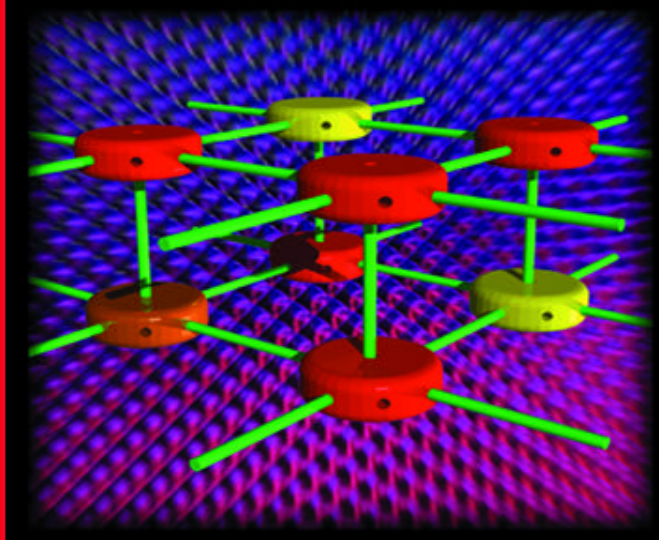
JAVA PROGRAMMING EXPLORER



The most complete guide to hands-on Java programming!

CD-ROM INCLUDED

- Over 50 applets ready to run
- Over 100 example programs with source code
- Full multimedia Java programs

- Presents extensive tutorials on how to create custom Java applets
- Provides ready-to-use Java templates created by Java pioneers

Neil Bartlett, Alex Leslie, and Steve Simkin

CORIOLIS GROUP BOOKS

III

Graphics: Learning to Sew with Sequins

Neil Bartlett

You've already learned how to paint. Now, you learn about drawing, colors, text, and fonts.

This chapter is all about drawing. That means that it is also all about the **Graphics** class.

The **Graphics** class is a marvelous class. It is the class that allows us do drawing. Using the **Graphics** class, we can indulge the "artist-within" and create any manner of free-form drawings. We can draw shapes, splash color onto a canvas, crop the picture, and indulge in some calligraphy.

If you have been following what we have been saying about object-oriented programming, you will appreciate that object-oriented tools such as the JDK do not give all the work to one class. The same is true of drawing. The **Graphics** class itself does not do all of the work. There is a whole host of support classes that the **Graphics** object will call upon to get the drawing job done. We will be hearing about the **Canvas** class, the **Point** class, the Polygon class, the **Font** class, the **Text** class, the **Color** class and even the **Random** class. All these and more will be called upon to help us do some drawing.

We have already met the **Graphics** class several times—notably in Chapter 13, the graphics and user interface tutorial. If the **Graphics** class is entirely unfamiliar to you, I recommend that you read that chapter before this one.

SEWING WITH SEQUINS

My wife and I live in Ontario, Canada. My wife, Mary, works in a private school in the town of Oakville. Mary, who insists I mention her youth and boundless energy, teaches younger children. Among other things, she teaches them arts and crafts.

Recently, she decided that she wanted to teach her children how to sew designs onto fabric using sequins. She figured that sewing sequins would be good for their dexterity, while the pictures they formed would be helpful in teaching them about art.

Mary decided to practice a bit herself before she introduced her students to their new artform. It was not long before I started to notice little packets of sequins appearing around the house—and it was not long after that we had more sequins hidden in the couch than we had lost coins.

One evening Mary came home and started telling me about the difficulties some of the children were having when they tried to turn their hand-drawn designs into refrigerator-ready sequin art. Their hand-drawn designs looked good on paper, but when the kids tried to create them with the sequins, the designs became unrecognizable.

It was at this point that I decided to tell Mary that she was also teaching her children how to program graphics on a computer. Like the true computer-geek husband, I was delighted that Mary at last wanted to use my computer skills to help her, and it was not long before I was launching into a "here's the problem" spiel. For once, this was not met by a polite "yes, dear" smile, but by a look of genuine interest. Soon we had out sheets of graph paper and were looking at how to make shapes like circles appear circular when composed entirely of a grid of dots.

The fabric that is used for sequin sewing consists of a rectangular array of holes which, if you draw the parallel, is rather similar to the grid of pixels that make up a computer screen, as shown in Figure 15.1.

Back to Java. If you want an analogy, the pixels are sequins and the **Graphics** class is like the needle and thread that sews the sequins to the fabric. The component classes are like the fabric that the **Graphics** class sews the pixel to. There

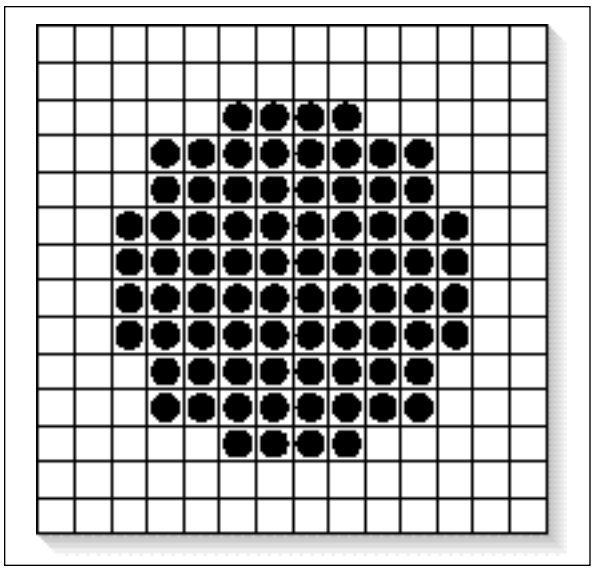


Figure 15.1 *A circle made of sequins.*

is plenty of fabric to choose from. The best quality fabric is the **Canvas** class, but the applet and the frame window also make good fabrics. Most of the component and container classes can be pushed into service as a fabric if need be.

THE COORDINATE SYSTEM

Objects of the "fabric" classes such as a **Canvas** object will arrange for an area of a window to be left clear, ready for a Graphics object to draw some graphics into it.

To determine where to place the things it wants to draw, the **Graphics** class uses a drawing grid called the *coordinate system* to position things in the window. The coordinate system is like the grid of the sequin fabric. It is a rectangular grid of points. Each point in the coordinate system is numbered. It takes two numbers to specify each point—the horizontal number and the vertical number. The horizontal number is called the x-coordinate and is the number of points the current point is from the left of the grid. The vertical number is called the y-coordinate and is the number of points the current point is from the top of the screen. This means that the origin of the coordinate system—the point labeled (0,0) is at the top-left of the grid. If you took (and, one hopes, passed) middle-school algebra, then Figure 15.2 should be quite familiar.

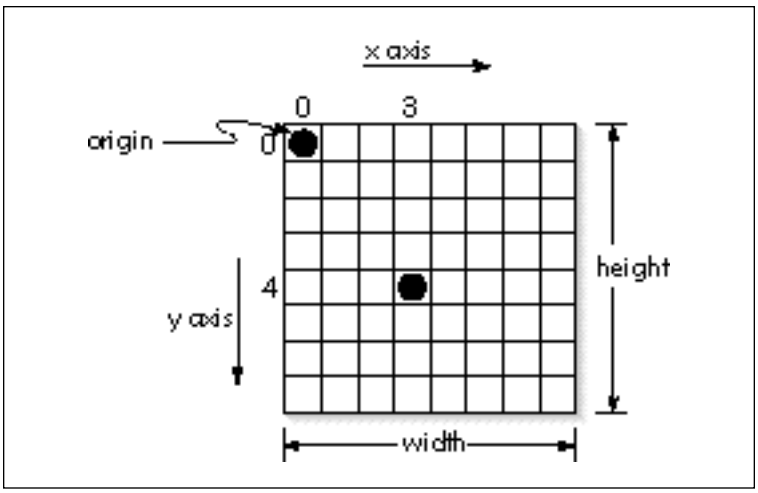


Figure 15.2 *The coordinate system.*

At school, you might have been used to drawing charts and graphs—maybe in math or physics. In these charts, the origin (0,0) was invariably at either the center or the bottom-left of the paper. It was never at the top-left of the paper. Computer graphics systems vary, but it is common to find the top-left corner as the origin of the system.

The Programs in this Section

All of the programs in this section are contained on the CD-ROM under the \SOURCE\CHAP15 directory. If you are interested in compiling and running these programs, you should first copy the programs to a directory on your hard disk, and then use the javac command to compile them. For example, to compile the ShapeTest.java program, which we'll see in a moment, you can do the following:

```
cmd> cd your-directory
cmd> javac ShapeTest.java
```

As you will be aware, there are several ways to run a java program, depending on whether the program is an applet or a standalone program. All of the techniques shown in this chapter can be applied to either type of Java program. Most sample programs shown in this chapter are standalone programs. You can run them using the java command. For instance, to run the ShapeTest program, use the following code line:

```
cmd java ShapeTest
```

You will find applet versions of some of the programs. The applet versions are named by appending the word "Applet" to the end of the program name. Naturally, applet files will need HTML files to run them—they are supplied as the *.html* file with the same "name" as the applet. The code snippet below shows an example of compiling and running the applet version of the shape test program:

```
cmd> javac ShapeTestapplet.java
cmd appletviewet ShapeTestapplet.html
```

LINE DRAWING: A RECAP

Before we go too far into this chapter, let's recap what we know about graphics drawing. We will look again at drawing a line on a canvas.

We first need to obtain a **Graphics** object. The **Graphics** object provides us with methods to do the drawing. Unfortunately, **Graphics** objects are not that forthcoming. The main way to get hold of one is via the **paint** method. The **paint** method is called each time the **Canvas** class must be redrawn.

To do our drawing, we code up the **paint** method to do the drawing we want to do.

Lines are possibly the simplest things that can be drawn using a **Graphics** object. To draw a line, we need to provide two points: the start point and the end point. We can draw lines using the **drawLine** method provided by the **Graphics** object. It takes the start and the end points as two sets of two numbers. In other words, we use four numbers (the x and y of the start point, and the x and y of the end point) to describe the line. For example, Listing 15.1 is a **Canvas** object with a **paint** method which will draw a line from the point (10, 20) to the point (190, 170).

Listing 15.1 Drawing a Line

```
class LineCanvas extends Canvas {
    public void paint(Graphics g) {
        g.drawLine(10, 20, 190, 170);
    }
}
```

Naturally, the **Canvas** object will need to be created inside of a program to be useful. As we have seen, we have a number of ways to set 11up the user

interface program: we can use an applet, a frame window, or even a frame window from inside of an applet. In Listing 15.2, we use the **Canvas** object from Listing 15.1 in an applet. Listing 15.3 shows the HTML to drive the applet. Figure 15.3 shows the applet.

Listing 15.2 LineCanvasApplet.java

```
import java.awt.*;
import java.Applet.*;

class LineCanvas extends Canvas {
    LineCanvas() {
       resize(200, 200);
    }

    public void paint(Graphics g) {
        g.drawLine(10, 20, 190, 170);
    }
}

public class LineCanvasApplet extends Applet {
    public void init() {
        add( new LineCanvas() );
    }
}
```

Listing 15.3 linecanvas.html

```
<applet code="LineCanvasApplet.class" width=200 height=200>
</applet>
```

The LineCanvasApplet.java file draws the line into a **Canvas**. However, the canvas is contained inside of an applet.

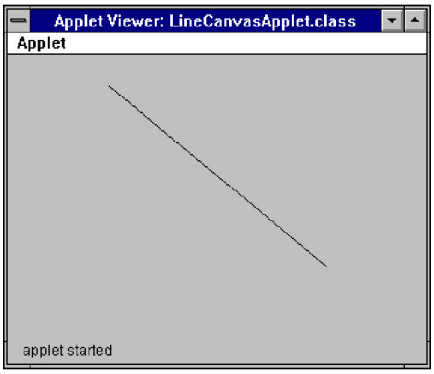


Figure 15.3 *LineCanvasApplet.java.*

FABRICS TO DRAW ON

The best class to draw on is the Canvas class. It has the very best support for drawing and it is likely to remain the best for drawing.

Other classes, such as those for the applet and frame window, are also good for drawing on. However, you must remember that these classes are also containers. Containers, as we saw in the last chapter, can be used to lay out other components inside of the area that the container has been allocated. Mixing containers' layout and drawing is not necessarily a very good idea. The layout manager in the container will lay out the components in potentially different places for different screen sizes.

When using the applet or the frame window as drawing areas, remember not to add other components into them. For instance, do not use code like that shown in Listing 15.4.

Listing 15.4 Bad Mix of Container and Drawing

```
public class BadDrawApplet extends Applet {
    public void init() {
        add( new Button("Press Me") );
    }
    public void paint(Graphics g) {
        g.drawLine(10, 20, 190, 170);
    }
}
```

Here the **Applet** class is being used as a place to draw on and as a container for a button. As you can see from Figure 15.4, this will produce some results, but they will not be consistent or reliable. If you want to mix components and drawing, it is far better to use a **Graphics** object, as illustrated in Listing 15.2.

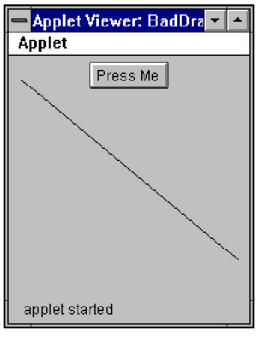


Figure 15.4 *Mixing components and drawing.*

However, if you do not intend to use the applet as a container, it makes as suitable place to draw on. Listing 15.5 shows drawing directly onto an applet. The result is shown in Figure 15.5.

Listing 15.5 LineDrawApplet.java—A Direct Drawing Applet

```
import java.awt.*;
import java.Applet.*;

public class LineDrawApplet extends Applet {

    public void paint(Graphics g) {
        g.drawLine(10, 20, 190, 170);
    }
}
```

METHOD DRAWING

You cannot create a **Graphics** object yourself. You have to be given one at runtime. If you take the time to browse through the JDK AWT source, you will find that there are very few places where you can get hold of a **Graphics** object. The best place to get a **Graphics** object is to have one passed into a **paint** method of a suitable drawing object.

There are other ways of getting a **Graphics** objects. There is the **update** method of any component or container, and then there is the **getGraphics** method of the **Image** class. We will be looking at these methods later. In general, however, the best place to do drawing is in the **paint** method.

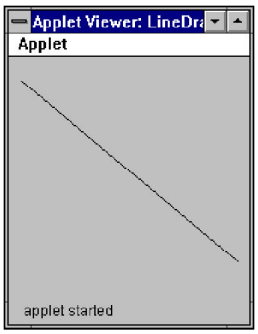


Figure 15.5 *Direct drawing Applet.*

Using the Point Class

The AWT provides us with a **Point** class. It is fairly bare bones. It provides a way of wrapping up the x and y coordinates of the point. You might be wondering, if you have looked at the **Graphics** class, why the **Graphics** methods do not use the **Point** class to specify points. You might also wonder why there is a **Point** class at all.

In fact, the **Point** class is used mainly as a class to return the x and y coordinates from a method. Recall that Java does not directly provide pass by reference; it is not possible to pass in a parameter to a method and get back the changed value in the same parameter. So you *can't* say things like

```
void method(int x, int y) {...}
```

where the method changes the values of x and y, and gets back the changed values in x and y. For this reason, **Point** objects are used as return values from these type of methods. So we could rewrite the above method as:

```
Point method(int x, int y) {... }
```

The method returns one thing: the **Point** object. The changed values of the x and y values would be returned in the **Point** object.

Conversely, to save us from having to construct unnecessary **Point** objects, the **Graphics** methods all take x and y coordinate parameters rather than **Point** objects.

A COMPENDIUM OF GRAPHICS OBJECTS

So that we can visualize the facilities provided by the **Graphics** class, let's put together a Graphics drawing compendium. It does for the methods of the **Graphics** class what the **PeriodicTable** did for the subclasses of the **Component** class. The directory for the compendium code is in the \SOURCE\CHAP15 directory. Refer to the instructions at the beginning of this chapter for running the programs.

Figure 15.6 shows the ShapeTest program running. The program consists of four rectangles (a drawn rectangle, two 3D rectangles, and a round-cornered rectangle), two polygons (one star and one filled star), a circle, an ellipse, and an arc. Some of these, you will notice, are not perfectly formed. We will be discussing why shortly.

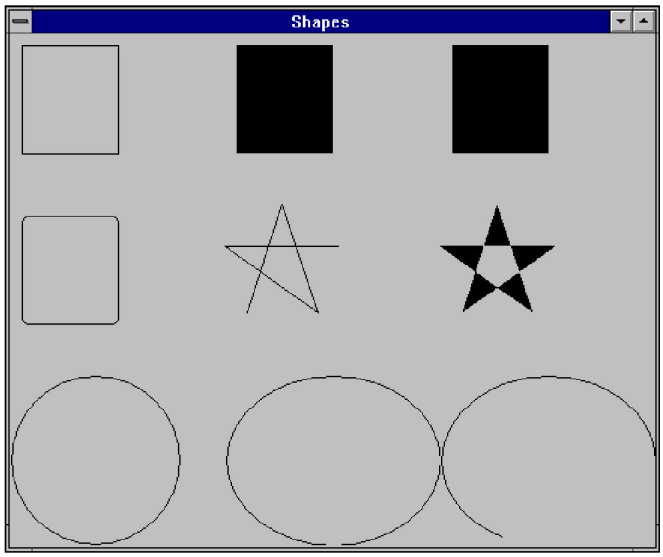


Figure 15.6 *The ShapeTest program.*

Listing 15.6 shows the ShapeTest program source code listing. The program is arranged as a grid of canvases inside of a frame window. Each canvas displays a different type of drawing object. There are rectangles, polygons, ovals, and arcs. To do the drawing, the **paint** method of each **Canvas** object is overridden and the appropriate **Graphics** method called to draw the required shape.

Listing 15.6 ShapeTest.java

```
import java.awt.*;
import java.lang.*;
import java.util.*;

class RoundRectCanvas extends Canvas {
    public void paint(Graphics g) {
        g.drawRoundRect(10, 10, 80, 90, 10, 10);
    }
}

class RectCanvas extends Canvas {
    public void paint(Graphics g) {
        g.drawRect(10, 10, 80, 90);
    }
}

class Rect3DCanvas extends Canvas {
    boolean raised;
```

```
    Rect3DCanvas(boolean b) {
        raised = b;
    }
    public void paint(Graphics g) {
        g.fill3DRect(10, 10, 80, 90, raised);
    }
}

class CircleCanvas extends Canvas {
    public void paint(Graphics g) {
        Dimension d = size();
        int diameter = Math.min(d.width, d.height)-2;
        g.drawOval(1, 1, diameter, diameter);
    }
}

class EllipseCanvas extends Canvas {
    public void paint(Graphics g) {
        Dimension d = size();
        g.drawOval(1, 1, d.width-1, d.height-1);
    }
}

class ArcCanvas extends Canvas {
    public void paint(Graphics g) {
        Dimension d = size();
        g.drawArc(1, 1, d.width-1, d.height-1, 0, 245);
    }
}

class PolygonCanvas extends Canvas {

    public void paint(Graphics g) {
        Polygon poly = new Polygon();
        poly.addPoint(18, 90);
        poly.addPoint(47, 0);
        poly.addPoint(77, 90);
        poly.addPoint(0, 35);
        poly.addPoint(95, 35);
        g.drawPolygon(poly);
    }
}

class FillPolygonCanvas extends Canvas {

    public void paint(Graphics g) {
        Polygon poly = new Polygon();
        poly.addPoint(18, 90);
        poly.addPoint(47, 0);
        poly.addPoint(77, 90);
        poly.addPoint(0, 35);
        poly.addPoint(95, 35);
        g.fillPolygon(poly);
    }
}
```

```
public class ShapeTest extends frame {

    public ShapeTest() {

        super("Shapes");

        setLayout( new GridLayout(3, 3) );

        add( new RectCanvas());
        add( new Rect3DCanvas(true));
        add( new Rect3DCanvas(false));
        add( new RoundRectCanvas());
        add( new PolygonCanvas());
        add( new FillPolygonCanvas());
        add( new CircleCanvas());
        add( new EllipseCanvas());
        add( new ArcCanvas());

        resize(550, 550);
        show();
    }

    public static void main(String args[]) {
   frame f = new ShapeTest();
    }
}
```

In the next few sections, we look at the **Graphics** drawing methods, shown in the ShapeTest program.

The Bounding Box

One thing that you will notice in common with most of the **Graphics** drawing methods is that they use the concept of a bounding box to specify the shape. The bounding box is a rectangle that encloses the shape to be drawn. For example, a circle can be drawn by using the **drawOval** method and using a square as the bounding box. The **drawOval** method will draw a circle so that the circle just touches the bounding box at the top, right, left ,and bottom.

The bounding box is specified using three facts: the upper-left corner of the rectangle, the width, and the height of the rectangle. The upper-left corner is used to position the rectangle and the width and the height are used to size the rectangle, as shown in Figure 15.7.

When supplying the bounding box as the parameter to the drawing methods, the JDK does not create a bounding box object. Instead, it requires the x and y coordinates of the bounding box and the width and the height. Each is passed as a separate parameter. The main reason for doing this is so as not to

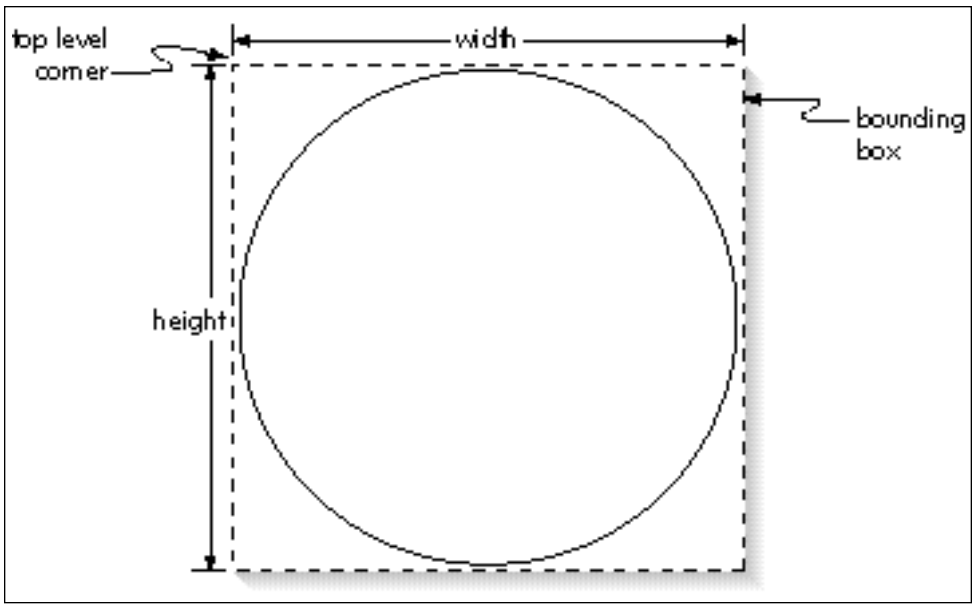


Figure 15.7 *A circle with its bounding box.*

incur the overhead of creating a separate bounding box object. This will give better performance when drawing.

The use of the bounding box may seem a little odd at first, especially if you are more accustomed to drawing circles on a piece of paper using compass needles or a coin. However, once you get used to it, it does seem very natural. Also, the bounding box concept has uses beyond the drawing of shapes. It is an important concept in many areas of graphics. We will return more to this subject later.

Rectangles

There are three types of rectangles that the **Graphics** object provides for us: rectangle, rounded rectangle, and 3D rectangle. Listing 15.7 shows a **paint** method that will draw a "regular" rectangle. The "regular" rectangle is drawn as an unfilled box.

Listing 15.7 An Unfilled Rectangle

```
class RectCanvas extends Canvas {
    public void paint(Graphics g) {
        Dimension d = size();
        g.drawRect(0, 0, d.width, d.height);
```

```
    }
}
```

All the rectangles are drawn using the bounding box for their size and position. The straight rectangles need no more information to specify them fully. The rounded and the 3D rectangles, though, are a little more complicated and will need some explanation.

Rounded Rectangles

Rounded rectangles are rectangles with rounded corners. The rounded rectangle is assumed to be equally rounded at each of the corners. You can imagine the rounded edges to be created by circles that are drawn at each of the four corners. The only difference is that the inner parts of the circles are not drawn. Only the parts that form a smooth curve between the sides of the rectangle are drawn. The corners do not have to be limited to circles. You can use ellipses if that is more the shape you need. An example is shown in Figure 15.8

Being able to use circles or ellipses means that you have full control over the amount of curve (by adjusting the size of the circle or the ellipse) and the shape of the curve (by adjusting the shape of the circle or ellipse).

Rounded rectangles need two extra facts to specify them: the width and height of the bounding box of the circle/ellipse. However, since the position of the bounding box is not needed—the drawing routines will draw them in the

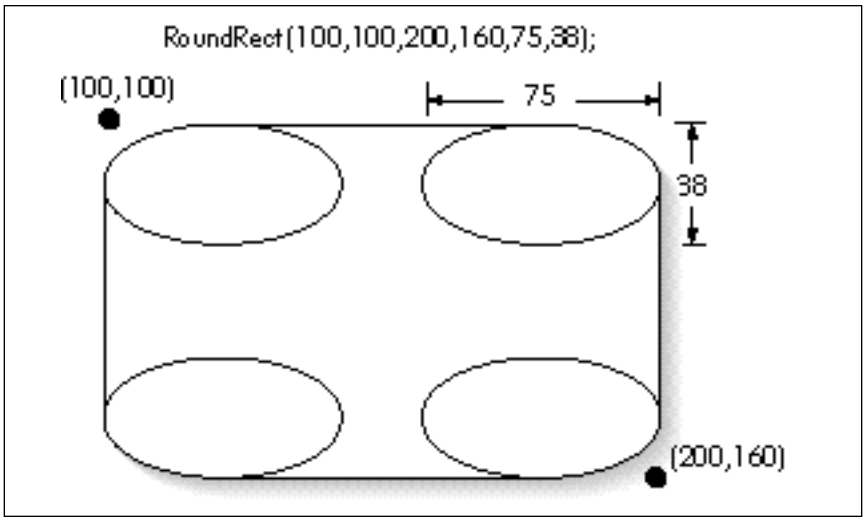


Figure 15.8 *A rounded rectangle with circles on its corners.*

correct positions in the corners—we need only specify the width and height of the bounding box.

In Listing 15.8, I chose to draw a rounded rectangle with rounded corners that are a tenth of the size of the rectangle's height and width.

Listing 15.8 Drawing a Rounded Rectangle

```
class RoundRectCanvas extends Canvas {
    public void paint(Graphics g) {
        Dimension d = size();
        g.drawRoundRect(0, 0, d.width, d.height,
                        d.width/10, d.height/10);
    }
}
```

3D Rectangles

3D rectangles are designed to give the impression of being either raised or lowered. This makes them useful for simulating buttons. 3D rectangles are best used as filled shapes rather than drawn shapes, though methods to draw both are provided. The 3D effect is not very strong in the drawn shape. Later we will write an improved version of the 3D rectangles. Listing 15.9 shows a sample piece of code to draw a 3D rectangle.

Listing 15.9 Drawing a 3D Rectangle

```
class Rect3DCanvas extends Canvas {
    boolean raised;

    Rect3DCanvas(boolean b) {
        raised = b;
    }
    public void paint(Graphics g) {
        Dimension d = size();
        g.fill3DRect(0, 0, d.width, d.height, raised);
    }

}
```

Another point to remember about 3D rectangles is the choice of colors. The algorithm that creates the 3D effect places darker or lighter colors at the edge of the rectangle to create the effect. However, because it is not possible to lighten white or to darken black. The effect will not look good for rectangles that are drawn in black or white—or colors close to them. For this reason, I generally choose **Color.lightGray** as the color when I am working with 3D rectangles. We will be looking at color management shortly.

Curved Shapes

The JDK has a number of methods for drawing curved shapes. The main method is a routine called **drawOval**. The JDK has decided to lump together all the circles and ellipse drawing routines and called them ovals. The circle is just an oval with a square bounding box. Listing 15.10 shows a canvas that will draw a circle.

Listing 15.10 Drawing a Circle

```
class CircleCanvas extends Canvas {
    public void paint(Graphics g) {
        Dimension d = size();
        int diameter = Math.min(d.width, d.height);
        g.drawOval(0, 0, diameter, diameter);
    }
}
```

Arcs

Arcs are provided by the **Graphics** object. They're interesting from several, pardon the expression, "points of view."

They are the only place in the AWT where angles are used. The angles used to specify arcs are measured in degrees. This contrasts with, say trigonometric methods in the **Math** class, which use radians as their way of specifying angles. The JDK does not provide a mechanism to convert between degrees and radians.

Arcs, as shown in Figure 15.9, are drawn using the **drawArc** method. It takes two parameters in addition to the bounding box: the starting angle and the

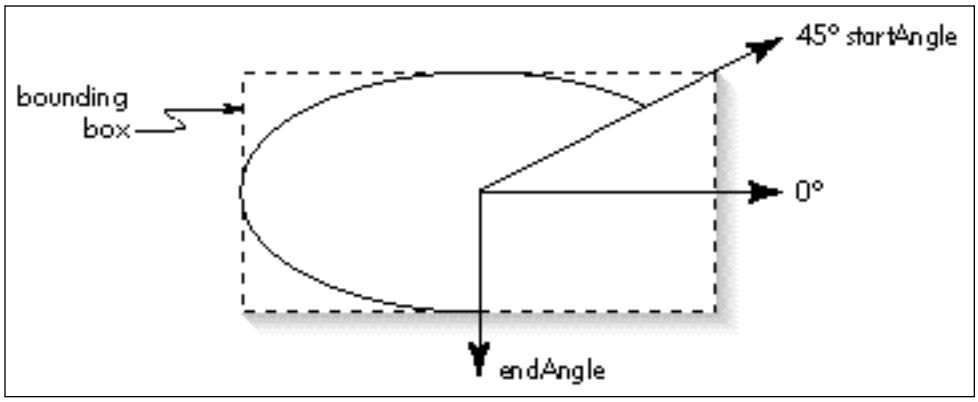


Figure 15.9 *The geometry of an arc.*

finishing angle of the arc. The starting and ending angles are measured starting from 0, which is at the three o'clock position (horizontally to the right). A positive number means a counter-clockwise rotation from three o'clock; a negative number means a clockwise rotation. Listing 15.11 draws the arc shape shown in the ShapeTest.

Listing 15.11 Drawing an Arc

```
class ArcCanvas extends Canvas {
    public void paint(Graphics g) {
        Dimension d = size();
        g.drawArc(0, 0, d.width, d.height, 0, 245);
    }
}
```

In addition to **drawArc**, there is a filled arc method, **fillArc**. This can be used to draw pie charts.

Polygons

In a book like this, I can't really resist the old joke:

Q: What shape is made by a parrot when it is squashed under the front wheel of a car?

A: A poly-gone.

Polygons, as you probably remember from school, are many sided shapes—the root word of "poly," of course, meaning "many" (in Greek) or "parrot" (in English). Here, we will only be concerned with the Greek meaning of the term.

The **Graphics** class allows us to describe a polygon using the points at the corners of the polygon. Polygon drawing departs from the bounding box mechanism of specifying shapes. There are two ways to specify the corners of the polygon. The first is to use two arrays of **ints**, the second is to use a **Polygon** object.

Using the array of **ints** technique is useful if we know how many points are in the polygon. We can then construct two arrays: one for the x coordinates and one for the y coordinates. For example, we can use

```
int xpoints[] = new int(5);
int ypoints[] = new int(5);
```

to construct an array ready for the points of a five-sided figure. This is not such a good technique if the number of points is unknown when the point

arrays are constructed, because you will need to reconstruct the size of the arrays as you fill them.

Using the **Polygon** class is very handy since the class manages the construction and dynamic sizing of the point array for you. The **Polygon** class is the most sensible mechanism to use when dynamically constructing a polygon, especially when you don't know how many points will be required. I tend to use the **Polygon** class technique most of the time because it is so much simpler to use.

An important concept about specifying polygons is whether they are closed or not. A closed polygon is a polygon like a square: one in which all the sides join up. However, as we will see, the **Graphics** polygon drawing methods will *not* close the polygon *for* us. If we want to have the polygon closed, we have to repeat the starting point at the end of the list of points.

Drawing a Polygon

The **drawPolygon** method draws the points of a polygon. It connects the points by drawing lines between the points. It starts with the first point, then draws a line from the first to the second, from the second to the third, and so on, until it reaches the last point.

The example we will use here, shown in Figure 15.10, almost draws a five-pointed star. To show that the **drawPolygon** method does not close the polygon, we will not add the duplicated first point. The code is shown in Listing 15.12.

Listing 15.12 A Non-Closed Polygon

```
class PolygonCanvas extends Canvas {

    public void paint(Graphics g) {
        Polygon poly = new Polygon();
        poly.addPoint(18, 90);
        poly.addPoint(47, 0);
        poly.addPoint(77, 90);
        poly.addPoint(0, 35);
        poly.addPoint(95, 35);
        g.drawPolygon(poly);
    }
}
```

Now let's look at the same shape through the eyes of the **fillPolygon** method. The **fillPolygon** method will fill the shape that we have produced. Listing 15.13 shows this.

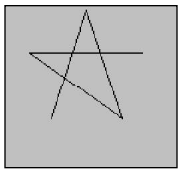


Figure 15.10 *A non-closed polygon.*

Listing 15.13 A Filled, Five-Point Star

```
class FillPolygonCanvas extends Canvas {

    public void paint(Graphics g) {
        Polygon poly = new Polygon();
        poly.addPoint(18, 90);
        poly.addPoint(47, 0);
        poly.addPoint(77, 90);
        poly.addPoint(0, 35);
        poly.addPoint(95, 35);
        g.fillPolygon(poly);
    }
}
```

If you look at Figure 15.11, you will notice that the star has a hollow center. This is because of the algorithm used to draw the polygon.

Polygons are filled by trying to determine if each point is inside or outside the polygon. If the point is inside, the area in which the point is contained is filled. Determining if a point is inside a polygon may seem intuitive to us humans, but the computer needs an algorithm to sort this out. The rule that decides if a point is inside of a polygon works like this: imagine a point inside of the polygon. Now draw a line out of the polygon from this point. Count the number of lines of the polygon that this escaping line crosses. If the escaping line crosses an odd number of lines then the area which contains the point is deemed to be inside of the polygon. If the count is even it is outside. This algorithm is shown in Figure 15.12.

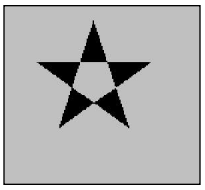


Figure 15.11 *A filled, five-point star.*

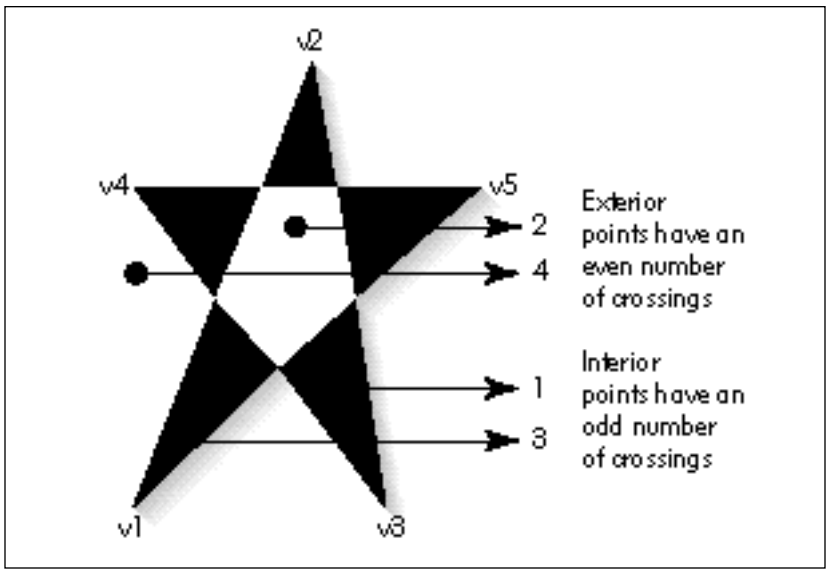


Figure 15.12 *The odd-parity rule.*

The reason this algorithm is used is that it runs fast. It works well for regular polygons such as squares, but crashes in pathological cases such as the five-point star. Some graphics systems, for example Windows, provide two polygon drawing algorithms that can be chosen by the programmer to correspond to the type of polygon being drawn. The **Graphics** class does not yet provide that degree of sophistication. Naturally, you could write your own, but we are talking very hardcore graphics here, so I wouldn't recommend it.

CREATING YOUR OWN SHAPES

The drawing methods we have looked at so far are the standard shape drawing methods provided by the **Graphics** objects. However, this does *not* mean that we are limited to these shapes. We are at liberty to construct our own shape drawing routines and use these in addition.

There are some notable examples of constructing shapes available on the Web. A good example is the Curve applet by Michael Heinrichs. This applet allows manipulation of various types of curves: Hermite, Bezier, and b-spline. It is available from **http://fas.sfu.ca:80/1/cs/people/GradStudents/heinrica/personal/curve.html**

Let's go on to look at doing some shape creation of our own.

Improving 3D Rectangles

I don't know if you noticed, but the 3D effect of 3D rectangles is not that strong. The 3D effect is simulated by placing a border around the rectangle (actually, just inside the rectangle). This border is one color around the top and left edges, and another color around the bottom and right edges. The colors are chosen so that for a raised effect the color is lighter on the top and darker on the bottom; and vice versa for a lowered effect. The idea is to imagine that there is a light placed at the top left of the screen. Like an artist shading in a picture to create a 3D effect, the colors are shaded around the box.

The problem with the current JDK implementation is that the border width is chosen to be very small—one pixel of the screen. This has two effects: the first is to minimize the 3D effect, while the second is to generate a harder edge.

The mechanism used by our improved algorithm is to draw the borders as two rectangles and two polygons. These two polygons would have been rectangles but to get a nice beveled edge effect they are sawn off at the corners. Figure 15.13 shows what is happening.

The colors are chosen using the brighter and darker methods that the **Color** class provides. Listing 15.14 shows the code for the improved 3D rectangle. To use this code, you will need to add the method to a class derived from, say, the **Canvas** class.

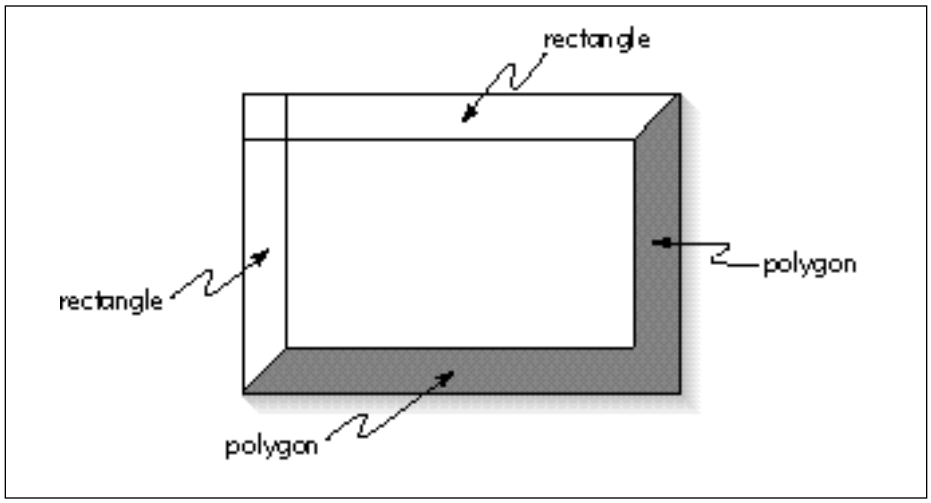


Figure 15.13 *Drawing the 3D rectangle.*

Listing 15.14 An Improved 3D Rectangle Drawing Method

```
public void
fill3DRect(Graphics g, int x, int y, int w, int h, int thick,
    boolean raised) {
    Color c = Color.lightGray;
    Color brighter = c.brighter();
    Color darker = c.darker();

    g.setColor(c);
    g.fillRect(x+thick, y+thick, w-(2*thick), h-(2*thick));
    g.setColor(raised ? brighter : darker);
    g.fillRect(x, y, w, thick);
    g.fillRect(x, y, thick, h);
    g.setColor(raised ? darker : brighter);

    int X[] = new int[4];
    int Y[] = new int[4];
    X[0] = x; Y[0] = y+h;
    X[1] = x+thick; Y[1] = y+h-thick;
    X[2] = x+w; Y[2] = y+h-thick;
    X[3] = x+w; Y[3] = y+h;
    g.fillPolygon(X, Y, 4);

    X[0] = x+w-thick; Y[0] = y+thick;
    X[1] = x+w; Y[1] = y;
    X[2] = x+w; Y[2] = y+h;
    X[3] = x+w-thick; Y[3] = y+h;
    g.fillPolygon(X, Y, 4);

    g.setColor(c);
}
```

Drawing Your Own Circles

This may seem like a particularly redundant thing to do, given that we can already draw good circles using the existing graphics routines. Writing our own routines, though, is useful as a way of learning how the system works. The programs in this section can be found in the \SOURCE\CHAP15\circle directory. Please refer to the beginning of the chapter for instructions on how to run them.

Okay, I admit what I am about to show you is a little hardcore. It is so hardcore that it might even be similar to an algorithm used deep down in the bowels of your operating systems graphics code, or possibly even hard-coded into the silicon on your graphics board.

In principle, the algorithm works like this. Imagine that you have a piece of graph paper. You stick a compass needle in the center of the circle and you

draw a circle on the graph paper. You want to represent the circle just using filled in squares, so you will need to carefully choose which squares to fill to make the best-looking circle. The algorithm proceeds by starting at one point on the circle, then moving to the next point by working out which square to hop to next.

Now we know that the x and y points on a circle are given by

$$radius^2 = x^2 + y^2$$

The cool thing about this algorithm—developed by a hot graphics expert called Bressenham in the 1970s—is that it does all its calculations using mainly adds and subtracts. It does not use squares and square roots, which are generally relatively slow on a computer. This makes the algorithm very fast. The algorithm I am using is taken from the book *Fundamentals of Computer Graphics* by Foley and Van Dam, second edition. The algorithm is in Chapter 3, pages 84-87. Listing 15.15 demonstrates the technique.

Listing 15.15 Drawing a Circle from Scratch

```
public static void drawPoint(Graphics g, int x, int y) {
    g.drawLine(x, y, x, y);
}

static void
circlePoints(Graphics g, int cx, int cy, int x, int y) {
    drawPoint(g, cx + x,  cy + y);
    drawPoint(g, cx + y,  cy + x);
    drawPoint(g, cx + y,  cy -x);
    drawPoint(g, cx + x,  cy -y);
    drawPoint(g, cx -x,   cy -y);
    drawPoint(g, cx -y,   cy -x);
    drawPoint(g, cx -y,   cy + x);
    drawPoint(g, cx -x,   cy + y);
}

static void
fillCirclePoints(Graphics g, int cx, int cy, int x, int y) {
    g.drawLine(cx - x,  cy + y, cx + x,  cy + y);
    g.drawLine(cx - y,  cy + x, cx + y,  cy + x);
    g.drawLine(cx - y,  cy - x, cx + y,  cy - x);
    g.drawLine(cx - x,  cy - y, cx + x,  cy - y);
}

public static void
drawCircle(Graphics g, int cx, int cy, int r) {
    int x = 0;
    int y = r;
```

```
    int d = 1 - r;
    int deltaE = 3;
    int deltaSE = -2*r+5;
    circlePoints(g, cx, cy, x, y);

    while( y>x ) {
        if( d<0 ) {
             d += deltaE;
             deltaE += 2;
             deltaSE +=  2;
             x = x + 1;
        } else {
             d += deltaSE;
             deltaE += 2;
             deltaSE += 4;
             x += 1;
             y -= 1;
        }
        circlePoints(g, cx, cy, x, y);
    }
}

public static void
fillCircle(Graphics g, int cx, int cy, int r) {
    int x = 0;
    int y = r;
    int d = 1 - r;
    int deltaE = 3;
    int deltaSE = -2*r+5;
    fillCirclePoints(g, cx, cy, x, y);

    while( y>x ) {
        if( d<0 ) {
             d += deltaE;
             deltaE += 2;
             deltaSE +=  2;
             x = x + 1;
        } else {
             d += deltaSE;
             deltaE += 2;
             deltaSE += 4;
             x += 1;
             y -= 1;
        }
        fillCirclePoints(g, cx, cy, x, y);
    }
}
```

I have written these methods so that they do not use the bounding box description mechanism that the other **Graphics** routines use. This makes these circle drawing routines more useful when we need to position a circle based on the center position. For instance, putting a centered circle at the end of a line is much easier using these circle methods.

One point to notice is that the **Graphics** class does not have a method to draw points (at least not yet, anyway). To simulate this, I have used a **drawLine** method with identical starting and end points. Another, and possibly better way, might have been to add the points into a **Polygon** object and then draw the object. This approach would have had additional performance advantages for drawing the filled circles.

The performance of these methods is slower but at least comparable on my machine to the native circle routines. Quite impressive. The Java code—a relatively high-level piece of code—is keeping up with code written with more machine-optimized methods.

Keeping Your Shapes Safe

A minor issue of writing your own graphics shapes is that you will probably want to use them in many circumstances, so there is the issue of how to package them. Obviously, it makes sense to put them in a central package such as our explorer package. However, how do we wrap up the method call?

There are three main options:

Option 1

We could write an extension of the **Graphics** class.

```
package explorer;
public class ExGraphics extends Graphics
```

This is not a particularly good option, because the **Graphics** class is an abstract class. This would mean that we would have to implement all of the **Graphics** methods or at the very least work out who would implement them.

Option 2

We could create a class which we initialize with a **Graphics** object. We would store this object inside the **ExGraphics** object. This has the advantage that we can override typical **Graphics** object methods by simply substituting the **ExGraphics** object for the **Graphics** object. We don't need to change the signature if the signature of our method is the same.

```
package explorer;

public class ExGraphics {
    Graphics graphics;
    ExGraphics(Graphics g) { ... }
}
```

The disadvantage is that we need to construct a separate **ExGraphics** object. If we are going to do a lot of graphics work with the **ExGraphics** object, then this could become an overhead.

Option 3
The third option is to define each new graphics shape-drawing method as static. These methods take an extra parameter, a **Graphics** object, which they will use to do the drawing.

This has the advantage that there is no extra construction involved. The **ExGraphics** simply becomes a collection of methods that can be called anytime without constructing a separate object.

```
package explorer;

public class ExGraphics {
    static drawFunc(Graphicsg, .... )
}
```

This last option is my personal preference. It does not suffer the potential overhead problems, though it does have the disadvantage of requiring the extra **Graphics** object in the parameter call.

CLIPPING

When a program draws lines, it can potentially draw very long lines: lines that could reach outside of the canvas. If these lines were completely drawn, then there could potentially be all kinds of interference between different components in a window. To stop this anti-social graphics behavior, a concept called *clipping* is used.

Clipping, as its name suggests, is the process that restricts the lines to make sure that they only display in the area where they are allowed. Clipping is normally done by using a *clipping rectangle*. All the lines and shapes are clipped so that only those portions that are in the rectangle are drawn, as shown in Figure 15.14. In this figure, only the solid black sections will be drawn the dotted sections will not be drawn.

When the AWT constructs a **Graphics** object to allow drawing to be done, it sets a clipping rectangle in the **Graphics** object, so that drawing done by the **Graphics** object will not affect other components in the same window.

This clipping rectangle is open for manipulation, using the **setClipRect** method. For instance, by reducing the size of the clipping rectangle, we can control

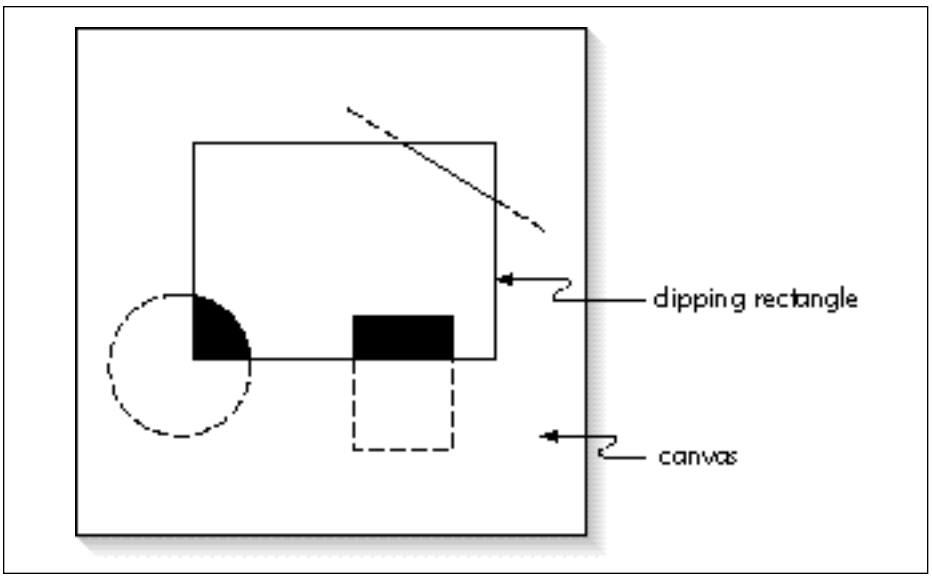


Figure 15.14 *Clipping in action.*

how much of the drawing we do actually gets to the canvas. This can be very useful when we want to do fast damage repair. For instance, if we are moving a rectangle around the screen, we might want to make sure that we only redraw the areas where the rectangle has moved from and which it has moved. This will probably result in faster and smoother movement of the rectangle.

CLEARING THE GRAPHICS

If you've played around a little bit with the canvas and graphics stuff, you've probably noticed that you do not have to clear the canvas before you do a paint. The act of repainting seems to clear the canvas to the background color.

This occurs because the canvas automatically draws the background just prior to doing a paint. The method that does the background clearing is the **update** method. The **update** method is also the method that calls the **paint** method. The purpose of separating the **update** and the **paint** method is precisely to do things that should happen before each paint, such as repaint the background.

If you don't want the background to be cleared each time, you can override the background drawing by overriding the **update** method with the following code:

```
public void update(Graphics g) {
    paint(g);
}
```

If you want to clear the graphics screen yourself, you use the **clearRect** method, which draws a rectangle in the background color. Here is some code to do that:

```
Dimension d = size();
g.clearRect(0, 0, d.width, d.height);
```

Transparent Backgrounds

There is a problem with the AWT if you want to have a transparent background. The problem is simple: you can't do it.

For instance, you might want to place an applet over a home page with a background image on it. You would like the home page background to show through as the background to the applet. In other words, you want to create a transparent applet background.

Currently, the background is drawn as a solid rectangle when the applet is first drawn. There is no way from the JDK to control this behavior, even by removing the "background" rectangle draw from the **update** method. Removing the background rectangle from the **update** does have other useful effects, which we will deal with later.

DRAWING MODES

The **Graphics** class provides two different types of drawing mode: the paint mode and the XOR mode.

Drawing mode refers to the way that the new shapes or lines being drawn interact with the things that have already been drawn. In paint mode, the new shape obliterates what was there before. The pixels that make up the shape are colored in the colors given by the new shape.

In XOR mode, this does not happen. The shape is drawn in a color that is dependent on the color of the shape to be drawn and the color of the shape that is already there. Where the shapes do not overlap, the shapes will be drawn in their own colors, but where they overlap a third color

will be used to draw the overlap. The third color will be generated by XORing the pixel values of the colors.

Typically, when doing drawing, paint mode is the most common mode to use. However, XOR mode can be useful when moving objects around. We won't be using XOR mode in this book.

CHARTING YOUR PROGRESS

As an example of what we can do with our newfound knowledge, let's look at a program to draw graphs and charts. As a side effect, we will create a program to help us visualize some of the math functions provided by the JDK and we can experiment with some biological randomness.

The programs in this section can be found in the \SOURCE\CHAP15\graph directory. Please refer to the beginning of the chapter for instructions on how to run them.

The source for the graph program is rather large. It is stored in the file GraphTest.java. We will fragment it into three parts and discuss each of the parts individually. First, we will discuss the **Graph** class, which stores the points for the graph. Then, we will discuss the **GraphPlotter** class, which draws the graph. Finally, we will take a look at a class that tests the other two classes to make sure they work correctly.

The Graph Class

At the heart of our graphing program is a simple class called **Graph**. It will maintain all of the points that we will be using to draw our graph. Listing 15.16 shows the source for the graph.

Listing 15.16 The Graph Class

```
class Graph extends Observable {

    double xminimum=Double.POSITIVE_INFINITY;
    double xmaximum=Double.NEGATIVE_INFINITY;
    double yminimum=Double.POSITIVE_INFINITY;
    double ymaximum=Double.NEGATIVE_INFINITY;

    /**
     * The total number of points
     */
    public int npoints = 0;

    /**
```

```
   * the array of x coordinates
   */
  public double xpoints[] = new double[10];

  /**
   * the array of y coordinates
   */
  public double ypoints[] = new double[10];

  Graph() {}

  /**
   * appends a point to the graph
   * @param x the x coordinate of the point
   * @param y the y coordinate of the point
   */
  public void addPoint(double x, double y)  {
if (npoints == xpoints.length) {
    double tmp[];

    tmp = new double[npoints * 2];
    System.arraycopy(xpoints, 0, tmp, 0, npoints);
    xpoints = tmp;

    tmp = new double[npoints * 2];
    System.arraycopy(ypoints, 0, tmp, 0, npoints);
    ypoints = tmp;
}
xpoints[npoints] = x;
ypoints[npoints] = y;
npoints++;

     xminimum = Math.min(xminimum, x);
     xmaximum = Math.max(xmaximum, x);
     yminimum = Math.min(yminimum, y);
     ymaximum = Math.max(ymaximum, y);
  }

  double X(int i) {
      return xpoints[i];
  }

  double Y(int i) {
      return ypoints[i];
  }

  int numPoints() {
     return npoints;
  }

  public double xmin() {
      return xminimum;
  }

  public double xmax() {
```

```
        return xmaximum;
    }

    public double ymin() {
        return yminimum;
    }

    public double ymax() {
        return ymaximum;
    }

    public void plot() {
        setChanged();
        notifyObservers(this);
    }
}
```

The coordinates of the points are stored in separate arrays of doubles. The **Graph** class has an **addPoint** method which will add points one at a time onto the graph. The **addPoint** method handles the sizing of the coordinate arrays. If the arrays are not big enough to hold the points, the arrays are dynamically resized.

The code that handles the resizing is interesting. It is not possible in Java to dynamically resize an array. You need to create a new array of the new desired size and then copy the contents of the old array into the new array. This copy can be slow, so the JDK supplies an array copying function which is optimized to be as fast as possible on each platform. This copying is supplied by the **System.arraycopy** method. The array copy method takes parameters of the two arrays and the indices in the array to copy from/to plus the number of array elements to copy. The **System.arraycopy** method will detect array overflow and types mismatches between the two arrays.

The **Graph** class stores the model of the graph—the abstract information we want to store. For this reason, we have made the **Graph** an *observable*. We talk more about **Observable** classes later in the section on Color. The **Observable** class means that other objects can register their interest in a **Graph** object and so be told when the **Graph** object changes.

Don't Hard-Code Numbers

Notice that we use the **Double.POSITIVE_INFINITY** and **Double.NEGATIVE_INFINITY** variables rather than hard coding the numbers directly ourselves. Whenever you need a mathematical number or a limit number, such as a maximum, for one of the built-in in data types, first try looking in the **Math** class or the number wrapper classes, such as **Double** or **Integer**.

Plotting the Graph

The **Graph** class is visualized using the **GraphPlotter** class. This will plot the points of the graph on a **Canvas**. Listing 15.17 shows the code for the **GraphPlotter** class.

Listing 15.17 The GraphPlotter Class

```
class GraphPlotter extends Canvas implements Observer {
    Graph graph;
    Dimension area;

    GraphPlotter(Graph g) {
       graph = g;
       g.addObserver(this);
    }

    public void update(Observable o, Object arg) {
        repaint();
    }

    Point plotPoint( double x, double y) {
        // DBG System.out.println("X:"+x+" Y:"+y);
        int offset = 10;
        int height = area.height - offset;
        int width = area.width - offset;
        int xpos = offset;
        int ypos = height;
        double xscale = width/(graph.xmax() - graph.xmin());
        double yscale = height/(graph.ymax() - graph.ymin());
        int px = (int) Math.round((x-graph.xmin())*xscale + xpos);
        int py = (int) Math.round(ypos - (y-graph.ymin())*yscale);
        return new Point(px, py);
    }

    public void paint(Graphics g) {
        area = size();
        Polygon poly = new Polygon();
        for (int i=0; i < graph.numPoints(); ++i) {
            Point p = plotPoint(graph.X(i), graph.Y(i));
            // DBG System.out.println(p.toString());
            poly.addPoint(p.x, p.y);
        }
        g.drawPolygon(poly);
    }
}
```

The main thing the **GraphPlotter** has to do is to convert the coordinate system. Currently the coordinate system for the canvas has the origin in the upper-left corner with the y values increasing down the screen. We want the origin to be near the lower left and have y values that increase up the screen

not down. Also we want to scale the value of the graph so that we make good use of the available canvas space.

To do our conversion, we need to do two things: we need to scale the value and translate the point. To scale the value, we work out a ratio of how many points are required per graph unit. For instance, if we look at the horizontal value, we need to work out the difference between the smallest and the largest values in the x coordinates of the graph and divide this into the number of units that are in the width. Then we will have a number that we can multiply a value by to get the graph coordinate of the value.

The translation adjusts the point of the origin by adding some offset values. It also requires that we negate the y value so that it increases up the screen.

Once we have converted the points stored in the graph to their display positions, we can plot the graph. We plot the points here using the **drawPolygon** method. As I previously noted, the **drawPolygon** method does not automatically join up the first and last points, so in effect it is a *polyline* method. In other words, we can use the **drawPolygon** method to draw lots of linked lines rather than having to call **drawLine** to draw each line segment.

Testing the Graph

To put the graph through its paces lets us use it to visualize the **Math.sin** method. We will construct a simple frame-based interface and put the **GraphPlotter** at the center of a **BorderLayout** in the frame.

The **sin** function is plotted by incrementing the angle passed to the **sin** function from -2Pi to +2Pi. Remember that the **sin** function takes radians, so this is equivalent to two entire sweeps of the circle as 2Pi radians is the same as 360 degrees. Listing 15.18 shows the test program.

Listing 15.18 Testing the Graph

```
public class GraphTest extends frame {

    void plotSin(Graph g) {
        for (double d = -2*Math.PI; d < 2*Math.PI/2; d+=0.1) {
            g.addPoint(d, Math.sin(d));
        }
    }
    public GraphTest() {

        super("Graph Plotter");

   setBackground(Color.lightGray);
```

```
        setLayout( new BorderLayout());

        Graph g = new Graph();
        plotSin(g);

        add("Center", new GraphPlotter(g));

        resize(550, 550);
        show();
    }

    public static void main(String a[]) {
        count  = (a.length >= 1) ? Integer.parseInt(a[0]) : 50000;
        numBars  = (a.length >= 2) ? Integer.parseInt(a[1]) : 100;
        frame f = new GraphTest();
    }
}
```

The program is a frame window which contains a **GraphPlotter**. The **GraphPlotter** is provided with a **Graph** object. The points to be plotted as added on the graph are added by the **Math.sin** method. The results of the program are shown in Figure 15.15.

A Biological Randomness

Now let's use our graph program to examine the output of the random number generator's **nextGaussian** method. The random number generator, which is provided by the **Random** class, provides a method called **nextGaussian**. This method is designed to provide a random number which conforms to the Gaussian normal distribution.

The Gaussian random number generator has a number of uses. It basically provides randomness near a number. The random number is more likely to

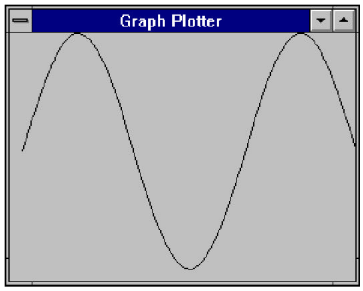


Figure 15.15 *Graph plot of the Math.sin method.*

be near the number that not near it. Let's run this through the graph and see what we get out. Listing 15.19 shows how to do it.

Listing 15.19 Plotting the Gaussian Distribution

```
public class GraphTest extends frame {
    static int count;
    static int numBars;

    void plotGaussian(Graph g, int n, int bars) {
        Random r = new Random();
        int values[] = new int[bars];
        for (int i= 0; i < n; ++i) {
            int value = (int) (bars/4*r.nextGaussian()+bars/2);
            value = Math.min(value, bars-1);
            value = Math.max(value, 0);
            values[value] += 1;
        }

        for (int i = 0; i < bars; ++i) {
            g.addPoint(i, values[i]);
        }
    }

    public GraphTest() {
        super("Graph Plotter");
        setBackground(Color.lightGray);
        setLayout( new BorderLayout());
        Graph g = new Graph();
        plotGaussian(g, count, numBars);
        add("Center", new GraphPlotter(g));
        resize(550, 550);
        show();
    }

    public static void main(String a[]) {
        count  = (a.length >= 1) ? Integer.parseInt(a[0]) : 50000;
        numBars  = (a.length >= 2) ? Integer.parseInt(a[1]) : 100;
        frame f = new GraphTest();
    }
}
```

The **plotGaussian** method takes the output of the **Random.nextGaussian** method and counts the number of times that the Gaussian produces a particular number. In other words, we are trying to produce a histogram or a bar chart of the number of times a particular random number is produced. This is shown in Figure 15.16.

If we called the **nextGaussian** method enough times, we should eventually see that the shape of the graph starts to look more and more like the Normal or **Gaussian** curve, as shown in Figure 15.17.

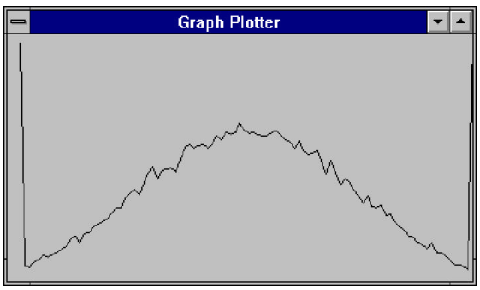


Figure 15.16 *Plotting the graph of the Gaussian distribution.*

So what's the big deal? For one thing, this Normal curve is very good at helping us simulate living things. The Normal curve has a tendency to a particular value—though it theoretically can be any value. Contrast this with other random numbers generated by the random number generator, which have equal likelihood of taking any number values.

A Better Graphing Program

The requirements I placed on our graph plotting program are rather arbitrary and restricting. I needed to produce particular types of graphs. As a conse-

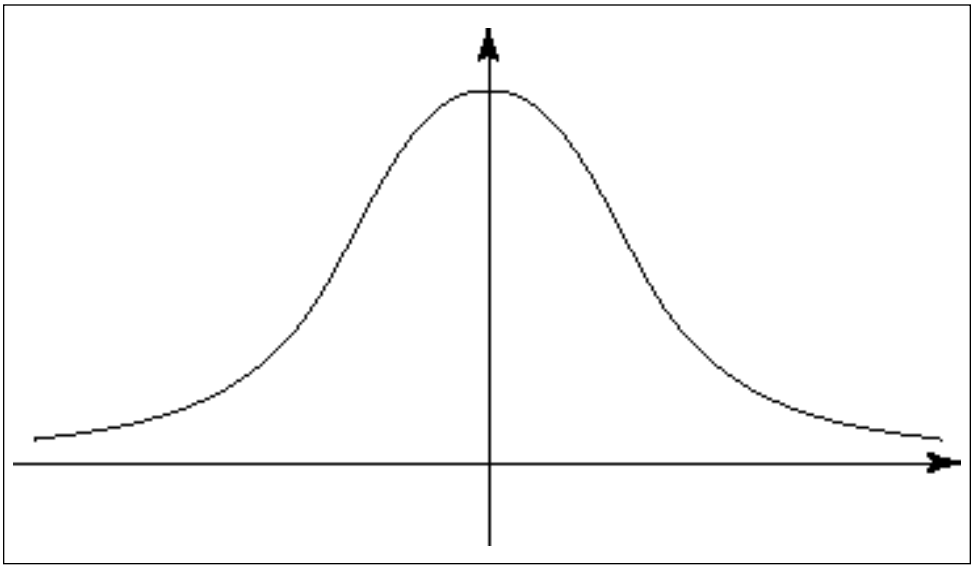


Figure 15.17 *Ideal Gaussian distribution.*

quence, the graphing program we have produced has lots of shortcomings: it plots graph shape rather than absolute values, which means that if the data is not ordered properly, a polygon will be drawn; and we have completely ignored cosmetics such as graph axes.

If you are after a good Java-based graphing package, there is one available on the Web at http://www-igpp.llnl.gov/people/brookshaw/java/. This package was written by Leigh Brookshaw. It has a lot of very interesting features and some of the test data that is displayed with the graph program is very interesting. I particularly like the volcano data. Take a look. I'm sure you will enjoy it.

THE RANDOM NUMBER GENERATOR

As we are here talking about parts of the random number generator, it is a good time to take a closer look at this beasty. The random number generator generates a sequence of random numbers. It can generate random number sequence for the following types found in Table 15.1.

To use the random number generator, you will need to construct an object of the **Random** class and call one of the methods shown in Table 15.1 to get a random number. Once the random number is constructed, you can use it to produces as many random numbers as you need by repeatedly calling any of the methods.

Restricting the Numbers

As you can see from the table, the Random class does not provide a way to generate a random number uniformly distributed over a range of values. To do this we need to do some work ourselves.

Table 15.1 *RandomNumber Generator Methods*

Method	Range Produced
nextInt()	Uniformly distributed over values possible from an int
nextLong()	Uniformly distributed over values possible from a long
nextFloat()	Uniformly distributed between 0.0 and 1.0; returned as a float
nextDouble()	Uniformly distributed between 0.0 and 1.0; returned as a double
nextGaussian()	Gaussian Normal distributed numbers with a mean of 0.0 and a standard deviation of 1.0; returned as a double

The simplest way to restrict numbers to a range of number is to use the modulus operator. Remember that the modulus operator provides the remainder when the number on the left is divided by the number on the right.

For instance, the code

```
result = x % 3
```

means that **result** can take the value 0, 1, or 2. The only proviso is that x must be greater than or equal to zero; otherwise, the modulus operator can't do its magic. To ensure that the value of x is not negative, we can use the **Math.abs** method.

A typical piece of code to get random numbers in the range 0 to n-1 is:

```
class someClass {
    Random r = new Random();

    int rand(int n) {
        return Math.abs(r.nextInt()) % n;
    }
}
```

Of course, there are more tricks we can play. For instance, if we want numbers in the range -(n-1) to (n-1), we will need to put back the sign information that the **Math.abs** method removed, as shown in this code snippet:

```
class someOtherClass {
    Random r = new Random();

    // assume that zero is a positive integer
    int sign(int i) {
        return (i <0) ? -1 : 1;
    }

    int rand(int n) {
        int i = r.nextInt();
        return sign(i) * Math.abs(i) % n;
    }
}
```

And that still leaves ranges like 1-5 to deal with! I'm sure you can cope with that though now.

A DICE PROGRAM

We can put the random number generator to good use to write a short dice program. The result of the program is shown in Figure 15.18. The program

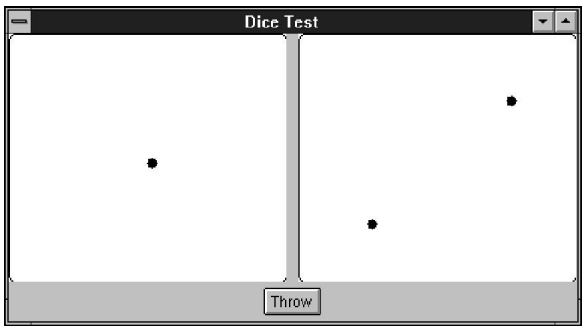


Figure 15.18 *Using the random number generator to create a dice program.*

consists of two die (yes, the plural of dice is die—isn't the English language wonderful) in a frame window. A button called Throw is also displayed. Each time the button is pressed, the die will randomly change their spots. The programs in this section can be found in the \SOURCE\CHAP15\dice directory. Please refer to the beginning of the chapter for instructions on how to run them.

How It Works

Listing 15.20 shows the complete source for the dice program. It consists of two classes: **DiceTest** and **Dice**. The **DiceTest** class constructs a frame window. The frame window is home to a panel that contains the two die and the Throw button. They are laid out using a **BorderLayout** class so that the button is placed at the bottom, the "South" of the window, while the Panel fills what is left of the window, the "Center." Each time the button is pressed, an action method is called. The action method will call the **throwDice** method for each of the dice.

Listing 15.20 DiceTest.java

```
import java.awt.*;
import java.util.*;
import java.lang.*;

public class DiceTest extends frame {
    Vector ds;

    public DiceTest() {

        super("Dice Test");
```

```
        setBackground(Color.lightGray);
        setLayout(new BorderLayout());
        Panel p = new Panel();
        p.setLayout( new GridLayout(1,2, 10, 10));
        ds = new Vector();
        for (int i = 0; i < 2; i++) {
            Dice d = new Dice();
            d.cheat(i);
            ds.addElement(d);
            p.add(d);
        }
        add("Center", p);
        p = new Panel();
        p.add( new Button("Throw") );
        add("South", p);
        resize(240, 140);
        show();
    }

    public boolean action(Event evt, Object obj) {
   if (evt.target instanceof Button) {
       String label = (String)obj;
       if (label.equals("Throw")) {
                for (Enumeration e=ds.elements();
                                 e.hasMoreElements(); ) {
                    Dice d = (Dice) e.nextElement();
                    d.throwDice();
                }
                repaint();
            }
   }
   return true;
    }

    public static void main(String args[]) {
   frame f = new DiceTest();
    }
}
class Dice extends Canvas {
    Random rand;
    int r;
    int value;

    Dice() {
        rand = new Random(hashCode());
        r = 10;
        setBackground(Color.white);
        setForeground(Color.black);
    }

    public synchronized void throwDice() {
        value = Math.abs(rand.nextInt()) % 6;
        // DBG System.out.println(toString());
        repaint();
```

```
    }

    void cheat(int n) {
        value = n % 6;
        repaint();
    }

    public int getNumber() {
        return value+1;
    }

    public String toString() {
        return new String("num="+value);
    }

    public void paint(Graphics g) {
        Dimension d = size();
        g.drawRoundRect(0, 0, d.width, d.height, 10, 10);
        switch ( value+1) {
            case 1:
                g.fillOval(d.width/2, d.height/2, r, r);
                break;
            case 2:
                g.fillOval(d.width/4, d.height*3/4, r, r);
                g.fillOval(d.width*3/4, d.height/4, r, r);
                break;
            case 3:
                g.fillOval(d.width/2, d.height/2, r, r);
                g.fillOval(d.width/4, d.height*3/4, r, r);
                g.fillOval(d.width*3/4, d.height/4, r, r);
                break;
            case 4:
                g.fillOval(d.width/4, d.height/4, r, r);
                g.fillOval(d.width*3/4, d.height/4, r, r);
                g.fillOval(d.width/4, d.height*3/4, r, r);
                g.fillOval(d.width*3/4, d.height*3/4, r, r);
                break;
            case 5:
                g.fillOval(d.width/2, d.height/2, r, r);
                g.fillOval(d.width/4, d.height/4, r, r);
                g.fillOval(d.width*3/4, d.height/4, r, r);
                g.fillOval(d.width/4, d.height*3/4, r, r);
                g.fillOval(d.width*3/4, d.height*3/4, r, r);
                break;
            case 6:
                g.fillOval(d.width/4, d.height/2, r, r);
                g.fillOval(d.width*3/4, d.height/2, r, r);
                g.fillOval(d.width/4, d.height/4, r, r);
                g.fillOval(d.width*3/4, d.height/4, r, r);
                g.fillOval(d.width/4, d.height*3/4, r, r);
                g.fillOval(d.width*3/4, d.height*3/4, r, r);
                break;
        }
    }
}
```

The heart of the dice program is the **Dice** class. This class is derived from the **Canvas** class. Each time the **throwDice** method is called, the dice will come up with a new number. When a **Dice** object is constructed, the dice will create a random number generator for itself. It will use this to determine how many spots to display on the dice. The **throwDice** method simple calls the random number generator to generate a new number, then calls a repaint. The **paint** method, which is called when the repaint happens, will draw the dice according to which number has been chosen by the random number generator. The **paint** method uses a **switch** statement to draw different patterns for each of the numbers. Each spot of the dice is drawn using a **fillOval** method set up to draw a circle.

Each dice class has its own **Random** object, which it uses to determine the dice value. As each dice has its own random number generator on-board, each dice should display an independent number. However, all is not as it appears. To find out more we need to understand a little more about random numbers.

Pseudo-Random Numbers

The numbers generated by the Java Random Number generator are what are termed *pseudo-random numbers*. Pseudo-random! How can a random number be pseudo? It sounds like the number is not random at all.

Actually, this is true. There is no way to produce a truly random number on a computer. The computer is what is termed a *deterministic* device. Deterministic means that if we do the same thing over and over again we always get the same result. If the computer were not deterministic it would not be so useful to us. After all, we want a device that will add two numbers up and for the same two numbers it will always give the same (hopefully, correct) answer.

However, this deterministic nature of the computer makes it difficult to produce a random number. After all, if it does the same thing over and over again—it is not random. To deal with this problem, pseudo-random numbers are produced.

Pseudo-random numbers are a long repeating cycle of numbers. The numbers are generated by a mathematical function that produces numbers appearing to be random. The pseudo-random numbers are *seeded* by an initial value. From there onwards, an algorithm will generate another number from the original number. The algorithm keeps generating pseudo-random numbers.

Eventually the numbers will cycle through and start over with the original value. Some random number generators have very long cycles—like 2 raised to the power 50—before they repeat the cycle. This does not mean that same

numbers do not repeat themselves during the 2 raised to the power 50 sequence. It means that a guy, sitting trying to guess what the next number was, would have to memorize 2 to the power 50 numbers before he would know for certain what the next one was. A tall order—even for the famed Russian memory man, S.

The "randomness" is heavily dependent on the *seed* value. If two random number generators are started with the same seed value, then the random number generators will generate the same sequence of numbers. This would not be much use to us with the **Dice** class we created earlier. If we had two dice and the random numbers were generating the same sequence, we would keep throwing doubles. This is handy in a game of snakes and ladders but not much use elsewhere.

To stop two random number sequences being the same, we must ensure that their seed numbers are different. If you just use the code

```
new Random()
```

to construct a random number, the **Random** class uses the current system millisecond time to seed the random number generator. This can cause problems in some cases. For instance, if we construct two dice, one after the other, chances are that the two could easily be constructed within the same millisecond because a computer can execute a large number of instructions inside a millisecond.

The random number generator does allow you to provide your own seed value. So what we need then is a unique seed number for each **Dice** object. There are probably many ways to do this. The technique I most commonly use is the **hashCode** method. The **hashCode** method is supplied by the **Object** class. This is the class from which all other classes inherit, so the **hashCode** method is available in classes. As the source code for Object.java says, this number is generally unique for each object, so it serves our purpose.

How the pseudo-random number generator actually works is well out of the scope of this book. If you are interested in finding out more, here are a few suggestions. The source code for Random.java recommends Donald Knuth's *The Art of Computer Programming, Volume 2* , Section 3.2.1. Donald Knuth has played a major role in the development of computer algorithms. His books are a very valuable contribution to computer science. However, a word of warning: they are serious and demand a lot of reading. If you'd like a more readable book, you might try *Algorithms* by Robert Sedgewick.

COLORS

Support for colors is provided via the **Color** class. It has a number of different methods to allow us to create different colors. The programs in this section can be found in the \SOURCE\CHAP15\COLEDIT directory. Please refer to the beginning of the chapter for instructions on how to run them.

There are two types of objects for which we can set colors. We can set colors for components such as the **Canvas**, and we can set colors in the **Graphics** object.

Components maintain the concept of a *foreground* and a *background* color. These can be manipulated using the set and get foreground and background color methods that are part of all component objects. The background color is the color that is used for the background of the window; the foreground color is that color that all drawing will be done in.

The **Graphics** object on the other hand only supports one color. The color is set using the **setColor** method and all drawing and filling is done using that color. When the graphics object is set up for the **paint** method, it is preset to be the foreground color of the component object that owns the **paint** method. Therefore, if all your drawing is to be in a particular color, you can just set the foreground color in the component. However, if you will be frequently changing the color you want to draw in, the best scheme is to change the color on the **Graphics** object.

Creating Colors

When we want a color we must get an object of the **Color** class. The **Color** class provides three ways for you to create colors.

Option 1: Standard Colors

The **Color** class lets us choose from a standard set of colors. The standard set of colors are specified in the **Color** class as static objects so there is no need to instantiate a new color. You can just use a standard color straight in any of the color setting routines. For example, to set the color blue as the current graphics drawing color in a **paint** method we might use:

```
public void paint(Graphics g) {
    g.setColor(Color.blue);
}
```

As you can see, we have not constructed the color object, we have used a straight value. There are a number of colors in the standard set. These colors 1are shown in Table 15.2.

Option 2: RGB Color Naming Scheme

RGB stands for Red, Green, and Blue. The RGB color scheme works by combining different amounts of red, green, or blue to form the color that you want. If you remember your color wheel from school, you will know that you can form any color if you combine the right proportions of red, green, and blue. Arguments abound as to whether white and black are colors, but even they can be formed using RGB. White is the color that is formed when large, equal amounts of red, green, and blue are mixed together; black is formed when none of the colors are present.

There are three constructors that allow you to create a color using the RGB system. The first two are shown in the following code lines:

```
public Color(int r, int g, int b)
public Color(float r, float g, float b)
```

These first two constructors take individual values for each of the red, green, and blue values. Using the **int** constructor, you can specify the values using any number between 0 and 255. 255 represents the color

Table 15.2 *Standard Colors*

Color	Red	Green	Blue
white	255	255	255
lightGray	192	192	192
gray	128	128	128
darkGray	64	64	64
black	0	0	0
red	255	0	0
pink	255	175	175
orange	255	200	0
yellow	255	255	0
green	0	255	0
magenta	255	0	255
cyan	0	255	255
blue	0	0	255

being fully present. In other words, you can add no more of that particular color. Blue, for instance, would be constructed as shown in the following code line:

```
new Color(0, 0, 255)
```

As you will have gathered, 0 means that there is none of a particular color present. In the blue constructor, there is obviously no red and no green present in the color.

The second version—the float version—of the constructor can be used to specify colors using float values between 0.0 and 1.0 instead of 0 and 255. A point to notice is that even though you can give finer grained numbers using floats, the float values do not give more accurate colors. The floats are first converted into the closest 0 to 255 **int** value before the color is created. The float version is given more for convenience than for better accuracy.

This final constructor is, in some sense, the most raw way to specify RGB. You can imagine in the code

```
public Color(int rgb)
```

that the single **int** value, which is composed of 32 bits, allocated 8 bits to each of the colors. If you know your binary arithmetic, you will know that 8 bits can be used to represent 256 different values: hence, the 0..255 range we have been using, as shown in Figure 15.19.

The RGB naming convention can be a little confusing at times. The concept of a color being completely present is kind of bogus. For instance, which has more blue: white (red=255, green=255, blue=255), yellow (red=0, green=255, blue=255) or blue (red=0, green=0, blue=255) ? Well, obviously, blue is bluer, so it must have more blue in it, but the RGB value for blue is the same in each case. The important thing to consider with RGB is the ratio of the colors, as well as the colors themselves.

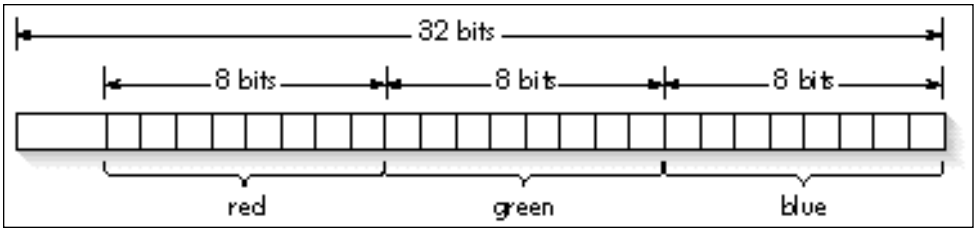


Figure 15.19 *Bit arrangement of an int RGB value.*

RGB is often represented as a cube, shown in Figure 15.20, with each of the colors taking a different axis of the cube. The color that you want is somewhere on the surface or on the inside of the cube.

Option 3: HSB Color Naming Scheme

HSB stands for Hue, Saturation, Brightness. It's another way of describing a color. Like RGB, the three HSB values can be used to describe any color. However, rather than break colors down in the three red, green, and blue colors, HSB uses a different mechanism.

The hue is the color or tint. Given a particular value for the hue, you would have a definite understanding of what the color was. Saturation is a measure of how strong the color is. It is a measure of the color purity. Brightness gives a measure of the lightness of the color. With a high brightness you can imagine a bright light being shone on the color, with a low brightness the color can be imagined as being viewed in a much darker setting.

The values for each of these numbers are floating point numbers between 0.0 and 1.0. Unlike the RGB values, the **Color** class does not provide a constructor; instead, a static method that will construct the color is provided, as shown in the code line:

```
public static Color getHSBColor(float h, float s, float b)
```

There are also several utility methods that allow conversion between the RGB naming scheme and the HSB naming scheme.

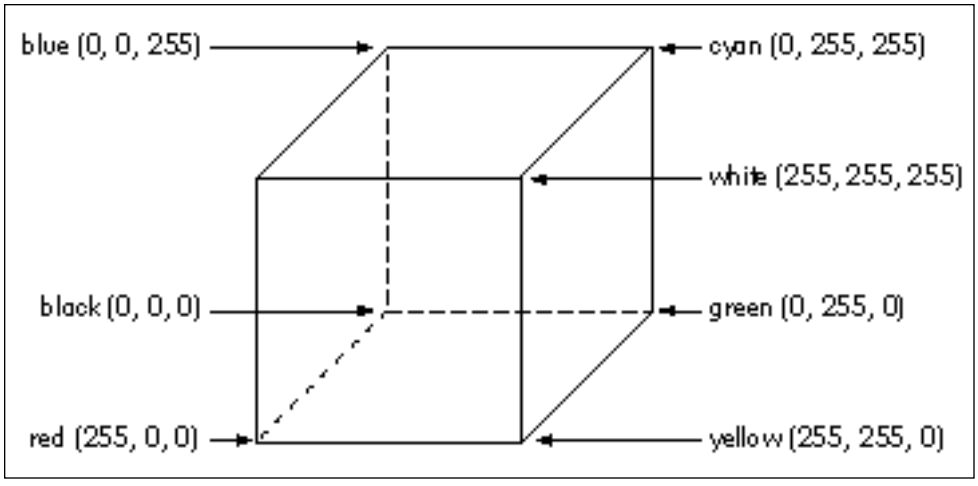


Figure 15.20 *RGB color cube.*

The HSB has a very interesting associated diagram, shown in Figure 15.21. RGB, we mentioned, can be viewed as a cube. HSB is typically imagined as a double cone with the points of the code being black and white. Like the RGB cube, the HSB color is either inside or on the surface of the cone.

These visualization techniques can come in very handy when you want to write color-selection programs. If you have done much work with good-quality painting programs you will probably recognize the visualization metaphor.

Why Three Ways to Create Colors?

So Java provides three ways to specify colors. One of them, the standard set of colors, is obviously provided as a convenience, but the other two seem to conflict. They both let us create a color using a combination of three values.

Having two color specification mechanisms seems like a certain amount of overkill. Surely one is enough. Well, sure, one *is* enough, but two are *very* useful. As it happens, there are numerous ways to specify colors in the computer world, but broadly speaking, they fall into two categories: those natural to the computer, and those useful to us humans.

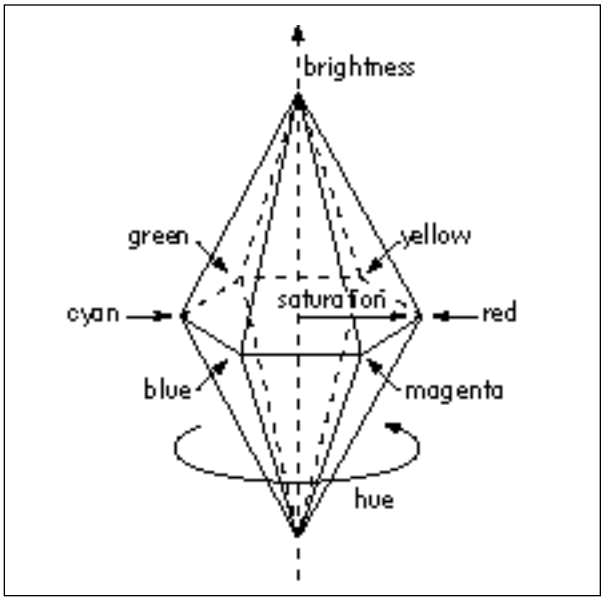


Figure 15.21 *HSB double cone.*

The **Color** class usefully provides one of each. The RGB format is the one closest to the computer way of using color. This probably comes as no surprise because TV sets use RGB guns to create a color picture. Most hardware graphics cards directly support the RGB format. You can load in the RGB value into a register in the card and the color is set.

The HSB value is much more in line with how we humans think.

Try this experiment: Below we present a program that lets us choose colors. Run the program, choose a color, then before changing, say, the brightness of the color, try to guess what the new color will look like. Try the same trick with RGB try to guess the outcome of changing one of the values. If you do this a number of times you will probably find that you can start to fairly accurately predict the new color using the HSB scheme, but using the RGB scheme it is much more difficult. Navigating your way around the RGB cube is a difficult process and one that takes a lot of learning. Moving around the HSB double cone is much easier—even though the double cone sounds like a tough thing to say and the cube sounds easier :-)

If you want to know more about color representation on computers, I recommend that you take a look at the book, *Fundamentals of Computer Graphics* by Foley and Van Dam. It is a truly heavy weight computer graphics book, but it will cover most everything you need to know!

Altering Colors

Once you have a color object, you can't change the value of the color that it represents. To get a new color, you need to create a new color object. You can get red, green, and blue values from an existing color object to help you create your new color, as shown in this code snippet:

```
Color color = g.getColor();
Color newColor = new Color(color.getRed(), color.getGreen(), 255);
```

In addition to creating new colors by calculating values yourself, the **Color** class also provides two convenience methods, **brighter** and **darker**, which will create new color objects that are either lighter or darker versions of the current color, as shown in this code snippet:

```
Color color = g.getColor();
Color brighterColor = color.brighter();
Color darkerColor = color.darker();
```

Color Your World

Let's write a little explorer program that will let us see what colors are produced by the different creation options. It allows you to create colors based on the three color selection mechanisms that the JDK supports: choosing a standard color, using RGB to create a color, or using HSB to create a color. It also provides features to alter the brightness and the darkness of the color. Figure 15.22 shows the program running.

The color in the center of the window can be changed using any of the four mechanisms provided.

At the bottom of the screen is a row of buttons. These buttons let you choose one of the standard colors. The standard colors are supplied as part of the **Color** class. When you press one of the buttons, the color will change to the selected color.

At the top of the screen are two buttons: Brighten and Darken. These buttons, as their names imply, Brighten and Darken the color that is currently displayed. You can keep darkening or lightening the color until eventually the color turns to black or white. Obviously, brightening white or darkening black makes no difference to the color.

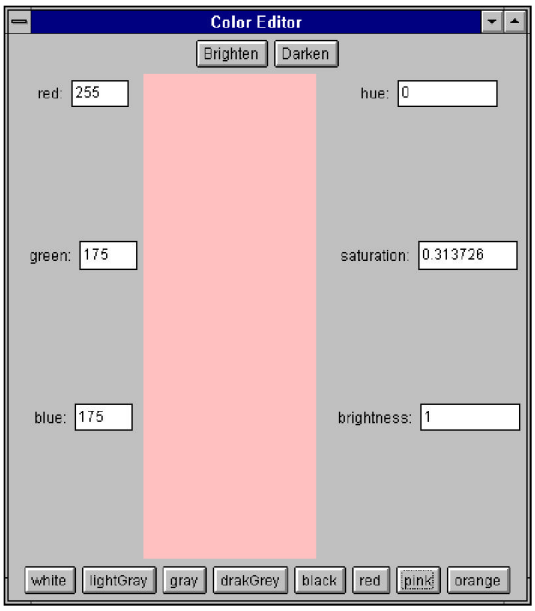


Figure 15.22 *The ColEdit program in action.*

The left side of the screen is used to set the color using the RGB color naming scheme. You will need to enter values between 0 and 255. The right side of the window is used to specify the color using the HSB. The values for each of these numbers are floating point numbers between 0.0 and 1.0. The code itself is shown in Listing 15.21.

Listing 15.21 ColEdit.java

```
import java.awt.*;
import java.util.*;
import extension.*;

class ColorCanvas extends Canvas implements Observer {
    ColorSetting colorSetting;

    ColorCanvas(ColorSetting s) {
        colorSetting = s;
        s.addObserver(this);
    }

    public void update(Observable o, Object arg) {
        repaint();
    }

    public void paint(Graphics g) {
        Dimension d = size();
        g.setPaintMode();
   g.setColor(colorSetting.getColor());
   g.fillRect(0, 0, d.width-1, d.height-1);
    }
}

class ColorSetting extends Observable {
    private Color col;

    ColorSetting() {
        col = new Color(0,0,0);
    }

    public void setRed(int r) {
        col = new Color(r, col.getGreen(), col.getBlue());
        tell();
    }

    public void setGreen(int g) {
        col = new Color(col.getRed(), g, col.getBlue());
        tell();
    }

    public void setBlue(int b) {
        col = new Color(col.getRed(), col.getGreen(), b);
```

```
        tell();
    }

    public void setColor(Color c) {
        col = c;
        tell();
    }

    public int getRGB(String c) {
        if (c.equals("red")) return col.getRed();
        if (c.equals("green")) return col.getGreen();
        if (c.equals("blue")) return col.getBlue();
        return 0;
    }

    public void setRGB(String c, int i) {
        if (c.equals("red")) setRed(i);
        if (c.equals("green")) setGreen(i);
        if (c.equals("blue")) setBlue(i);
        tell();
    }

    public float getHSB(String c) {
        if (c.equals("hue"))
            return Color.RGBtoHSB(col.getRed(), col.getGreen(),
                                  col.getBlue(), null)[0];
        if (c.equals("saturation"))
            return Color.RGBtoHSB(col.getRed(), col.getGreen(),
                                  col.getBlue(), null)[1];
        if (c.equals("brightness"))
            return Color.RGBtoHSB(col.getRed(), col.getGreen(),
                                  col.getBlue(), null)[2];
        return 0;
    }

    public void setHSB(String c, float f) {
        float hsb[] = Color.RGBtoHSB(col.getRed(), col.getGreen(),
                                     col.getBlue(), null);
        if (c.equals("hue"))
            col = Color.getHSBColor(f, hsb[1], hsb[2]);
        if (c.equals("saturation"))
            col = Color.getHSBColor(hsb[0], f, hsb[2]);
        if (c.equals("brightness"))
            col = Color.getHSBColor(hsb[0], hsb[1], f);
        tell();
    }

    public void brighten() {
        col = col.brighter();
        tell();
    }

    public void darken() {
```

```
        col = col.darker();
        tell();
    }

    void tell() {
        setChanged();
        notifyObservers(this);
    }

    public Color getColor() {
        return col;
    }
}

class HSBField extends Panel implements Observer {
    String name;
    ColorSetting colorSetting;
    TextField fld;

    HSBField(ColorSetting cs, String s) {
        name = s;
        colorSetting = cs;
        add( new Label(name+":"));
        add(fld = new TextField(Float.toString(cs.getHSB(s)), 9));
        cs.addObserver(this);
    }

    public void update(Observable o, Object arg) {
        ColorSetting cs = (ColorSetting) arg;
        String s = Float.toString( cs.getHSB(name));
        if (!s.equals(field.getText()))
            fld.setText(s);
    }

    public boolean action(Event evt, Object what) {
       float value = Float.valueOf(fld.getText()).floatValue();
       colorSetting.setHSB(name, value);
       return true;
    }
}

class RGBField extends Panel implements Observer {
    String name;
    ColorSetting colorSetting;
    TextField f;

    RGBField(ColorSetting cs, String s) {
        name = s;
        colorSetting = cs;
        add( new Label(name+":"));
        add(f = new TextField(Integer.toString(cs.getRGB(s)), 4));
        cs.addObserver(this);
    }
```

```
    public void update(Observable o, Object arg) {
        ColorSetting cs = (ColorSetting) arg;
        String s = Integer.toString( cs.getRGB(name));
        if (!s.equals(f.getText()))
            f.setText(s);
    }

    public boolean action(Event evt, Object what) {
        if (evt.target == f) {
            int value = Integer.parseInt(f.getText());
            colorSetting.setRGB(name, value);
            return true;
        }
        return true;
    }
}

class BrightenButton extends Button {
    ColorSetting colorSetting;

    BrightenButton(ColorSetting c) {
        super("Brighten");
        colorSetting = c;
    }

    public boolean action(Event evt, Object what) {
        if (evt.target == this) {
            colorSetting.brighten();
            return true;
        }
        return false;
    }
}

class DarkenButton extends Button {
    ColorSetting colorSetting;

    DarkenButton(ColorSetting c) {
        super("Darken");
        colorSetting = c;
    }

    public boolean action(Event evt, Object what) {
        if (evt.target == this) {
            colorSetting.darken();
            return true;
        }
        return false;
    }
}

class RGBButton extends Button {
    ColorSetting colorSetting;
```

```
    Color color;

    RGBButton(ColorSetting cs, String name, Color c) {
        super(name);
        colorSetting = cs;
        color = c;
    }

    public boolean action(Event evt, Object what) {
        if (evt.target == this) {
            colorSetting.setColor(color);
            return true;
        }
        return false;
    }
}

public class ColEdit extends frame {

    ColorSetting cs;

    public ColEdit() {

        super("Color Editor");

   setBackground(Color.lightGray);
        setLayout( new BorderLayout());

        cs = new ColorSetting();

        Panel p = new Panel();
        p.setLayout( new GridLayout(3,1));
        p.add(new RGBField(cs, "red"));
        p.add(new RGBField(cs, "green"));
        p.add(new RGBField(cs, "blue"));
        add("West", p);

        add("Center", new ColorCanvas(cs));

        p = new Panel();
        p.setLayout( new GridLayout(3, 1));
        p.add(new HSBField(cs, "hue"));
        p.add(new HSBField(cs, "saturation"));
        p.add(new HSBField(cs, "brightness"));
        add("East", p);

        p = new Panel();
        p.add(new BrightenButton(cs));
        p.add(new DarkenButton(cs));
        add("North", p);

        p = new Panel();
        p.add( new RGBButton(cs, "white", Color.white));
```

```
        p.add( new RGBButton(cs, "lightGray", Color.lightGray));
        p.add( new RGBButton(cs, "gray", Color.gray));
        p.add( new RGBButton(cs, "drakGrey", Color.darkGray));
        p.add( new RGBButton(cs, "black", Color.black));
        p.add( new RGBButton(cs, "red", Color.red));
        p.add( new RGBButton(cs, "pink", Color.pink));
        p.add( new RGBButton(cs, "orange", Color.orange));
        p.add( new RGBButton(cs, "yellow", Color.yellow));
        p.add( new RGBButton(cs, "green", Color.green));
        p.add( new RGBButton(cs, "magenta", Color.magenta));
        p.add( new RGBButton(cs, "cyan", Color.cyan));
        p.add( new RGBButton(cs, "blue", Color.blue));
        add("South", p);

        resize(550, 550);
        show();
    }

    public static void main(String args[]) {
   frame f = new ColEdit();
    }
}
```

As you can see from Listing 15.21, the program is based on a **Frame** object which contains all the buttons, text fields, and the canvas area that make up the program. To keep the program short, each of the RGB and each of the HSB fields are provided by the same component. The constructor decides which one to build. **RGBButtons** provides access to the standard colors. The Darken and Brighten buttons are formed by deriving buttons from the **Button** class.

The whole thing is laid out using a combination of panels each arranged into a **BorderLayout** format.

What will probably be confusing to you is the way that the program hangs together. There is a central class called **ColorSetting**, which seems to be the thing that all of the buttons and text fields talk to.

To understand how the program works, we need to look at a couple of other classes called **Observer** and **Observable**.

OBSERVER AND OBSERVABLE

Did you notice that the numbers in the HSB and the RGB fields changed when different colors were selected? Take a look. Whenever you press one of the color buttons, the values in the RGB and the HSB fields will change. Also,

if you change an RGB value, the HSB values will change to match the RGB, and vice versa.

It seems like the each of the different fields knows all about the other buttons and fields. For instance, if we press one of the standard colors buttons, it must set the values for the RGB and the HSB and set the color canvas to the correct color. Or when changing an HSB value, the HSB field must calculate the new RGB values and set the color in the center of the screen.

This sounds like a lot of work.

Actually the system does not work like this at all. I want to introduce you to two very useful and special classes: **Observer** and **Observable**. **Observer** and **Observable** are a great help in writing user interfaces on the screen that need to be kept instep with each other.

The idea is that the observable is managing some information that is being watched by the observers. When the observers want to watch an observable they register themselves with observable. They do this by calling the **addObserver** method of the **Observable** object, as shown in this code snippet.

```
s.addObserver(this);
```

The observable will maintain a list of all the observers that are watching it. When information held by the observable changes, the observable will call a method called **setChanged**, then a method called **notifyObservers**. This has the effect of calling an **update** method against each of the **Observer** objects, as shown in the code snippet below and in Figure 15.23.

```
setChanged();
notifyObservers(this);
```

The observers now know that the observable has changed and they are free to query the observable for more information.

Take the example we have here. We have a color that we are trying to change. We want to be able to alter that color, but we want all things that have an interest in the color to be told about the color.

You see, the color is something that is being changed. There are lots of things viewing this color though. The RGB fields are a view of the colors, as are the HSB fields, and the color canvas.

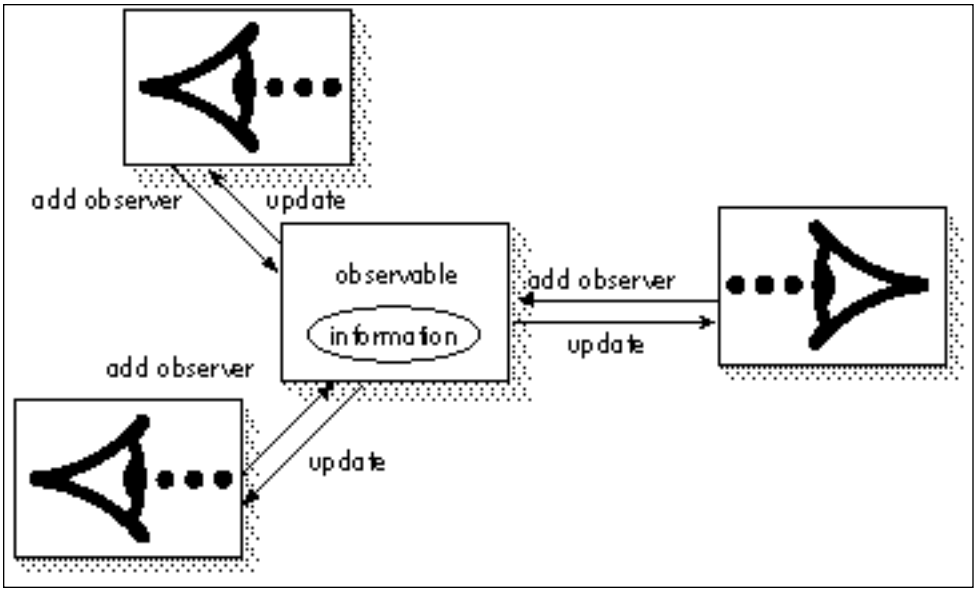


Figure 15.23 *The observer and observables.*

Confused? Well okay. A little confusion now and then is the hallmark of an active mind. Look at it this way: we have a concept called color; it is an abstract concept. We can't see the color, just a model of it. Now we can ask the color what its RGB values are and we can ask it what its HSB values are. We can also attach a view to the color. This color view will set the correct hardware values on our graphics card so we can see the color.

So the abstract color remains behind the scenes. We don't see it directly we only see it through the "eyes" of the RGB field, the HSB fields, and the color canvas. Take a look at the code back in Listing 15.21, the abstract color is represented by the class called **ColorSetting**.

The **ColorSetting** class extends the **Observable** class. In other words, it is being watched by other classes such as the **ColorCanvas** class.

The **ColorCanvas** class implements the **Observer** interface. That means that it must provide a method called **update**. When the **update** method is called the **ColorCanvas** knows that it observable, the **ColorSetting** object has changed, it can now update itself to be consistent with the new values stored in the **ColorSetting** object.

Take a good look at the two classes. Notice that the **ColorSetting** never calls the **ColorCanvas** class directly.

There are many uses for the **Observer** and **Observable** classes, but keeping the user interface up to date is probably the most common and the most useful. As an example of using **Observer** and **Observable,** take a look at the Visible Human applet, shown in Figure 15.24. This applet can be found at Web address http://www.npac.syr.edu/projects/vishuman/VisibleHuman.html.

The Visible Human applet shows a cross section of the human body at the top left of the screen. The cross section is governed by the two full views of the human body and by settings on the buttons and fields at the bottom of the screen. Given the interconnection between the various views and the various settings, how would you implement the Visible Human applet now that you know about the **Observer** and the **Observable** classes?

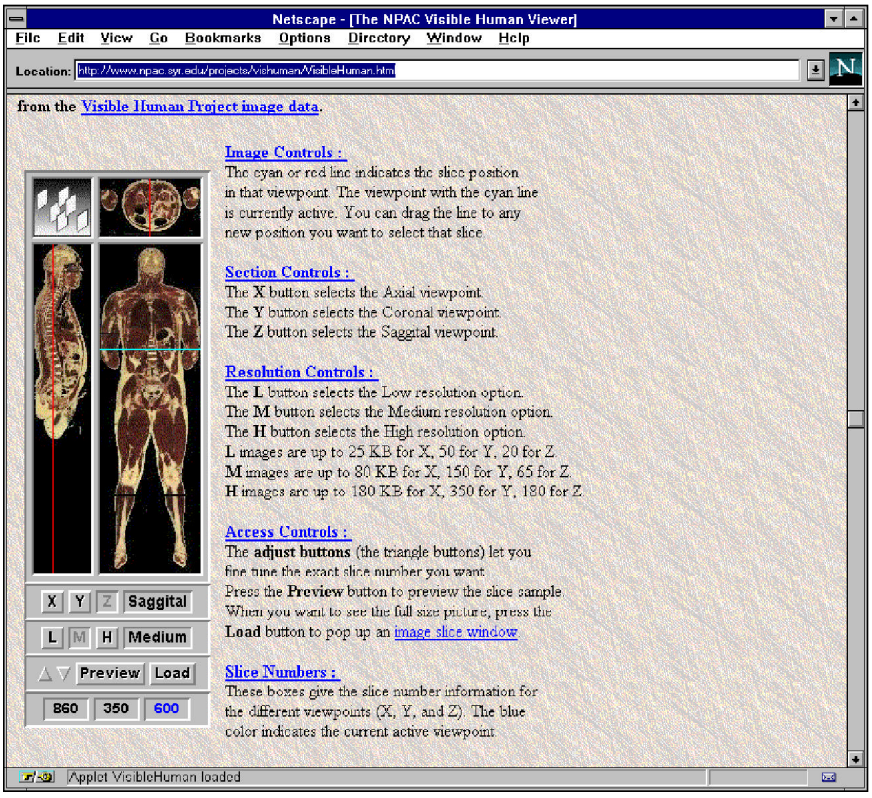

Netscape - [The NPAC Visible Human Viewer]
File Edit View Go Bookmarks Options Directory Window Help
Location: http://www.npac.syr.edu/projects/vishuman/VisibleHuman.htm
from the Visible Human Project image data.
Image Controls :
The cyan or red line indicates the slice position in that viewpoint. The viewpoint with the cyan line is currently active. You can drag the line to any new position you want to select that slice.
Section Controls :
The X button selects the Axial viewpoint.
The Y button selects the Coronal viewpoint.
The Z button selects the Saggital viewpoint.
Resolution Controls :
The L button selects the Low resolution option.
The M button selects the Medium resolution option.
The H button selects the High resolution option.
L images are up to 25 KB for X, 50 for Y, 20 for Z
M images are up to 80 KB for X, 150 for Y, 65 for Z
H images are up to 180 KB for X, 350 for Y, 180 for Z
Access Controls :
The adjust buttons (the triangle buttons) let you fine tune the exact slice number you want.
Press the Preview button to preview the slice sample. When you want to see the full size picture, press the Load button to pop up an image slice window.
Slice Numbers :
These boxes give the slice number information for the different viewpoints (X, Y, and Z). The blue color indicates the current active viewpoint.
X Y Z Saggital
L M H Medium
Preview Load
860 350 600
Applet VisibleHuman loaded

Figure 15.24 *The Visible Human applet.*

TEXT

The programs in this section can be found in the \SOURCE\CHAP15\TEXT directory. Please refer to the beginning of the chapter for instructions on how to run them.

Text can be drawn directly into the canvas using low-level text drawing methods provided by the **Graphics** class. These low-level text methods do not provide the support that a **TextArea** or a **TextField** component would provide. The **TextArea** component, for instance, knows about text lines and can be used to select text. The **Graphics** class text drawing methods do not support the newline mechanism. If the text runs off the edge of the canvas, it is clipped and will not be drawn: it does not start a new text line automatically.

The graphics text routines though have a lot of advantages. You can set the color of the text, you can control the text spaces, and best of all, you can write "Jiggly" text programs. "Jiggly" text programs are like a pet-rock fad among Java programmers. We will be looking at a "Jiggly" text program in a moment. Figure 15.25 shows a sample program that draws a few lines of text. It demonstrates how to draw text using the different drawing methods, and it shows how the newline character does not cause a newline on the screen. Listing 15.22 shows the code that produces the program.

Listing 15.22 TextTest.java

```
import java.awt.*;

class TextCanvas extends Canvas {
    Font font1, font2;

    TextCanvas() {
        font1 = new Font("TimesRoman", Font.PLAIN, 36);
        font2 = new Font("TimesRoman", Font.PLAIN, 24);
    }

    public void paint(Graphics g) {
        Dimension d = size();

        g.setFont(font1);
        g.setColor(Color.red);
        g.drawString("Hello", 20, 30);
        g.setFont(font2);
        g.setColor(Color.yellow);
        // see how the character does not cause a newline!
        g.drawString("Here is a string with a newline\nin it",
                        20, 60);

        g.setColor(Color.blue);
```

```
        FontMetrics fm = getFontMetrics(getFont());
        String str = "Stepup";
        int height = fm.getHeight();
        char[] chars = str.toCharArray();
        for (int i = 0; i < str.length(); i++) {
            int xwidth = fm.charsWidth(chars, 0, i+1);
            g.drawChars(chars, i, 1, 20+xwidth+1, 120-i*height/2);
        }
    }
}

public class TextTest extends frame {

    public TextTest() {
   setBackground(Color.lightGray);
        setLayout( new BorderLayout());
        add("Center", new TextCanvas());
        resize(550, 250);
        show();
    }

    public static void main(String args[]) {
   frame f = new TextTest();
    }
}
```

When text is drawn using the graphics text methods, you will need to specify the x and y coordinates of the baseline of the text. The baseline is like a line in a notebook: text without descenders sits on the line, characters with descenders such as g cross the line, as shown in Figure 15.26.

Currently, all individual characters are drawn in a left to right fashion. You cannot angle individual characters, though you can staircase the text to get a rising or descending text string. It is possible to rotate text, but it will require a lot of work on you part—you will need to draw the text into an image string then rotate the image string, then draw the image string in the correct place. There are a lot of issues involved here, so we will not cover this technique.

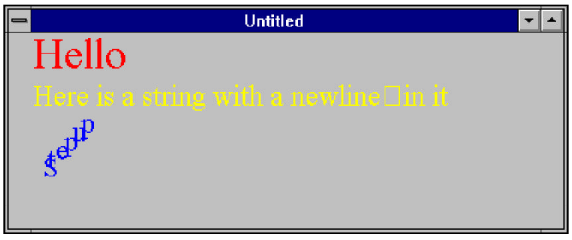


Figure 15.25 *The TestText program in action.*

Figure 15.26 *The baseline of text.*

FONTS

The programs in this section can be found in the \SOURCE\CHAP15\FONT directory. Please refer to the beginning of the chapter for instructions on how to run them.

The style and shape of the text that is drawn is controlled by the font of the text. There is always a default font available for text to be drawn in. The JDK provides a **Font** class which provides the main mechanism for describing fonts. You will need to construct a **Font** object if you want to change the font in use. Table 15.3 shows the attribute settings of the JDK fonts.

The font styles can be one or more of the following: Font.PLAIN, Font.BOLD, Font.ITALIC. To get a combination of the font styles, add together the styles. For example, to get a bold and italic font use the style shown in the following code snippet:

```
Font.BOLD+Font.ITALIC
```

Table 15.3 *Font Attributes*

Attribute	Set	Get	Description
Logical Name	setName	getName	The name of the font; the logical name is the same on all platforms
Family	setFamily	getFamily	Platform specific name of the font; this can be regarded as the font that 'implements' the logical font
Style	setStyle	getStyle	The style refers to the typeface of the font.
Size	setSize	getSize	The size of the font; the font size is given in point size

Listing All the Fonts Available on a Machine

A list of all the logical names available on a platform can be retrieved using the **Toolkit** class. The **Toolkit** class provides a method, **getFontList**, that will return a list of available logical font names as an array of Strings. We can get an object of the **Toolkit** class using the method shown in the code line:

```
Toolkit.getDefaultToolkit();
```

Listing 15.23 shows a short program to list the available fonts on to standard out.

Listing 15.23 Listing Available Fonts

```
import java.awt.*;
import java.lang.*;

public class FontList {

    public static void main(String args[]) {
   Toolkit tk = Toolkit.getDefaultToolkit();
        String[] fonts = tk.getFontList();
        for (int i=0; i < fonts.length; ++i)
            System.out.println(fonts[i]);
    }
}
```

When this program is run on my machine, I get the following list:

```
Cmdt>java FontList
Dialog
Helvetica
TimesRoman
Courier
Symbol
```

What Is a Toolkit?

In some sense, the **Toolkit** class is the creator class of the AWT. It is an interface to a lot of specific system information. For instance, we can get the screen size and a list of available fonts using the **Toolkit** class.

Most importantly, though, the **Toolkit** class is closely tied in with the peer classes we spoke about in a graphics tutorial chapter. The peer classes, you will recall, are the go-betweens that sit between the system-specific controls and AWT components. The **Toolkit** is the manufacturer of the peer classes. The **Toolkit** class is what is known as a factory class. In some sense, the **Toolkit** class is a stand-in for the operating system specifics. Each platform will have a different implementation of the **Toolkit** class, but because it pro-

vides the same interface on all platforms, the classes of the JDK can use the **Toolkit** class to get system-dependent things. For instance, components can use the **Toolkit** class to construct their corresponding peer class.

You might be tempted to ask: Why can't components just construct a peer class directly? The answer is that the **Toolkit** provides an extra degree of management. It can be used to ensure that system resources can be tuned. This enables the implementation of the platform specifics to retain a certain degree of control over the creation of system-specific resources.

Font Selection

Here is a simple program, demonstrated in Figure 15.27, to show the fonts that the JDK will let you use on your machine.

The idea of the program is that you can select a font using the three font description parameters: font name, style, and size. A string consisting of all the uppercase and lowercase characters in the font is drawn in that font. If the string will not fit on the screen, then as much of the string as possible will be shown. Below the string is a little information about the font.

When you take a look at the Font Selection program, be sure to try out different sizes and text styles. On my machine, this provided some very interesting information. Be careful how large a font size you choose. The program deliberately does not protect against large font sizes. I found on my machine that setting a large number—say 5000—as the point size was enough to hang my machine. This has a couple of impacts. First off, don't run the program when you have other programs with unsaved data. More importantly, you should consider providing limitations on the maximum size that can be set from programs that you write in which you allow your users to set the font size. If you do not, you will potentially run the risk of crashing user machines.

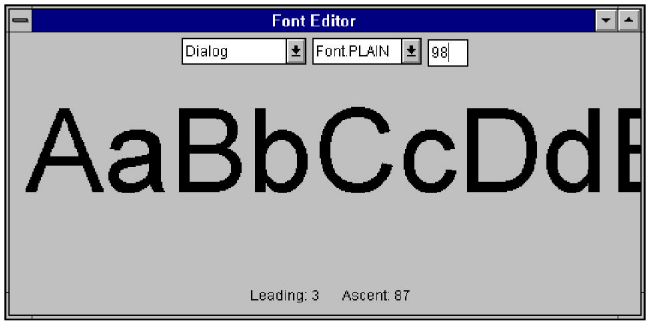


Figure 15.27 *The Font Selection program.*

Another aspect of setting the font size is so that you can see the effects of changing the size. Fonts generally come in two technologies—raster fonts and *vector* (or stroke) fonts.

Characters in raster fonts are just bitmaps which are drawn on to the screen. Raster fonts are fast to draw because it is fast to draw an image on the screen. However, raster fonts have a couple of draw backs. First, they are fixed size: a new raster font must be calculated for each new font size. Second, they require a lot of memory to hold all the bitmaps. A side effect of the raster fonts is that the description of the fonts is only done for certain point sizes. This means that in-between point sizes tend to be a bit jaggy, as shown in Figure 15.27.

Characters in vector fonts, on the other hand, are not stored as bitmaps. Instead, they are stored as a description—the outline is a list of points that must be joined up, like a connect-the-dots puzzle, and then filled in. Obviously, vector fonts are slower to draw than bitmap fonts because each character must be calculated however; they have the advantage of being stored as a single description and that one description can be used to describe any size of font.

When you experiment with the font size you will discover which fonts are raster and which are vector.

Implementing the Font Selector

The complete source for the FontSelector program is on the disk as the file FontSelector.java. The code is shown in Listing 15.24.

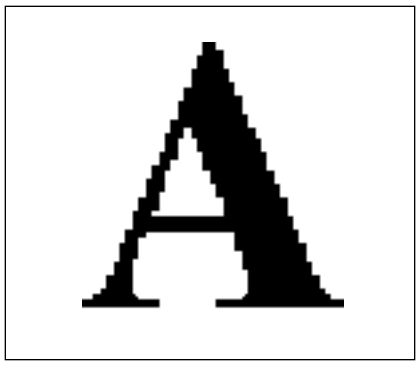


Figure 15.27 *An in-between point size of a raster font.*

Listing 15.24 FontSelector.java

```
import java.awt.*;
import java.util.*;
import extension.*;

class FontCanvas extends Canvas implements Observer {
    FontSetting fontSetting;
    int xoffset, yoffset;

    FontCanvas(FontSetting s) {
        fontSetting = s;
        s.addObserver(this);
        xoffset = yoffset = 10;
    }

    public void update(Observable o, Object arg) {
        // note setFont will cause a repaint
   setFont(fontSetting.getFont());
       // repaint()
    }

    public void paint(Graphics g) {
        Dimension d = size();
        g.setPaintMode();
        FontMetrics fm = getFontMetrics(getFont());
        g.drawString(
            "AaBbCcDdEeFfGgHhIiJjKkLlMmNnOoPpQqRrSsTtUuVvWwXxYyZz",
            fm.getMaxAdvance()+xoffset, fm.getAscent()+yoffset);
    }
}

class FontSetting extends Observable {
    private Font font;

    FontSetting() {
        font = new Font("Helvetica", Font.PLAIN,10);
    }

    public void setName(String s) {
        font = new Font(s, font.getStyle(), font.getSize());
        tell();
    }

    public void setType(String s) {
        if (s.equals("Font.PLAIN"))
            font = new Font(font.getName(), Font.PLAIN,
                            font.getSize());
        if (s.equals("Font.BOLD"))
            font = new Font(font.getName(), Font.BOLD,
                            font.getSize());
        if (s.equals("Font.ITALIC"))
            font = new Font(font.getName(), Font.ITALIC,
```

```
                            font.getSize());
        tell();
    }

    public void setSize(int p) {
        font = new Font(font.getName(), font.getStyle(), p);
        tell();
    }

    void tell() {
        setChanged();
        notifyObservers(this);
    }

    public Font getFont() {
        return font;
    }
}

class FontInfo extends Panel implements Observer {
    FontSetting fontSetting;
    Label leading;
    Label ascent;

    FontInfo(FontSetting fs) {
        fontSetting = fs;
        add( leading = new Label("Leading:          "));
        add( ascent = new Label("Ascent:            "));
        fs.addObserver(this);
    }

    public void update(Observable o, Object arg) {
        FontMetrics fm = getFontMetrics(fontSetting.getFont());
        leading.setText("Leading: "+fm.getLeading());
        ascent.setText("Ascent: "+fm.getAscent());
    }
}
class FontName extends Choice {
    FontSetting fontSetting;

    FontName(FontSetting fs) {
        fontSetting = fs;
   Toolkit tk = Toolkit.getDefaultToolkit();
        String[] fonts = tk.getFontList();
        for (int i=0; i < fonts.length; ++i)
            addItem(fonts[i]);
    }

    public boolean action(Event evt, Object what) {
        if (evt.target == this) {
            fontSetting.setName(getSelectedItem());
            return true;
        }
```

```
        return false;
    }
}

public class FontSelector extends frame {

    FontSetting fs;

    public FontSelector() {

        super("Font Editor");

   setBackground(Color.lightGray);
        setLayout( new BorderLayout());

        fs = new FontSetting();

        Panel p = new Panel();
        p.add(new FontName(fs));
        p.add(new FontType(fs));
        p.add(new FontSize(fs));
        add("North", p);

        add("Center", new FontCanvas(fs));

        add("South", new FontInfo(fs));

        resize(500, 750);
        show();
    }

    public static void main(String args[]) {
   frame f = new FontSelector();
    }
}
```

The FontSelector program uses a number of the techniques we have explored so far. The font is drawn using a **FontCanvas** class. The **FontCanvas** class is a descendant of the **Canvas** class, as well as an observer of a **FontSetting** class. When the **FontSetting** class changes, the **FontCanvas update** method is called. The method will then set the font for the **FontCanvas** object to the **FontSetting** object's Font. The font is set using the **setFont** method included with all component objects. The **setFont** class immediately forces a repaint. A word of warning here though: The forcing of the repaint is handled by the peer class. It is not coded into the **Component** class. This might mean that on your machine it may behave differently. I think that the coding of a forced repaint on a change of setting is a rather dubious behavior. If you find that your fonts don't change when you change the settings, I would first put in an extra repaint at the end of the **update** method.

The actual drawing of the string is handled in the **paint** method. It uses the **Graphics** method **drawString** to draw the upper- and lowercase string. The **Graphics** object uses the font for the component object it is created by. In this case the **Graphics** object uses the **FontCanvas** font object.

To ensure that the string is usefully visible, the font is positioned so that it is Ascent units from the top of the screen and Leading units from the left of the screen. To prevent the font from touching the sides of the screen, a 10 unit offset is also added in.

The **FontSetting** class is very similar in concept to the **ColorSetting** we used for the color program. It is an **Observable** object which stores the current setting of the font. As the font is changed it notifies all the **Observer** objects that have registered interest in the font settings that things have changed.

In this case, the only other **Observer** object beside the **FontCanvas** is the **FontInfo** object. This is the object that prints the font information on the current font into the scrolling text window. Each time the font settings are changed, the text window will replace its text with the information on the current font.

The changing of the setting is done by a couple of choice controls and a text field. These items simply call the appropriate method of the font settings object when they change. For example, below is the class for the **FontName** object, which sets the logical name of the font. This is a **Choice** component. The **Choice** component is filled using the **Toolkit** mechanism we saw earlier. When the choice is selected, the action event is called. The action event decodes the request to change the font name, gets the chosen name, and informs the font setting object.

The whole program is strung together using a combination of layout managers. The **FontCanvas** is set as the center of a **BorderLayout** to ensure that when a resize happens, it will get as much of the space as possible.

Font Metrics

If you want to know all the specific details about a font you will need to use the **FontMetrics** class. The **FontMetrics** class can access a lot of interesting information about the font, as shown in Figure 15.28. The methods that can be used to access this information are shown in Table 15.4.

The **FontMetrics** class is an abstract class. It cannot be instantiated. To get hold of a **FontMetrics** object you will need to use the **getFontMetrics** method

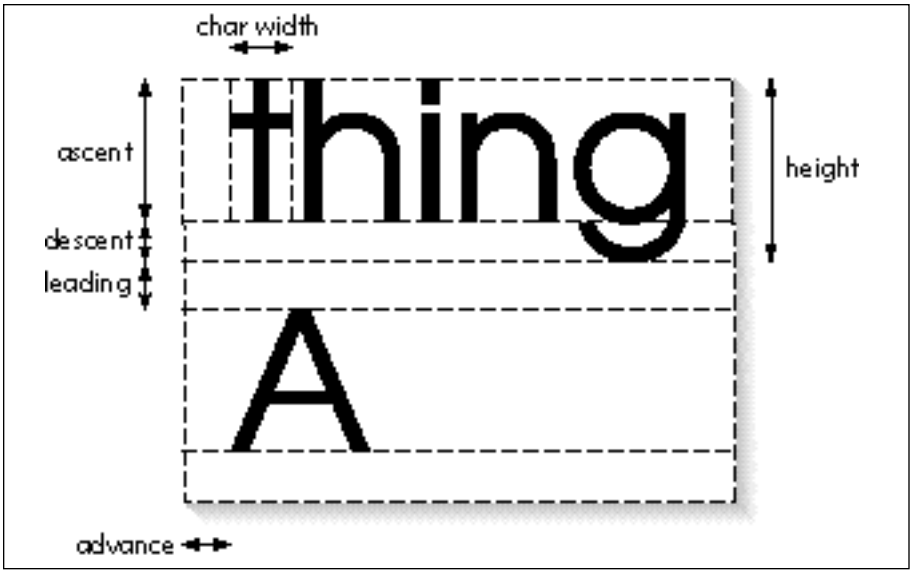


Figure 15.28 *Font metric information for a string.*

provided by component objects. For instance, to get the font metrics for the current font in, say, a **Canvas** object we can use the code shown:

```
FontMetrics fm = getFontMetrics(getFont());
```

Writing Text That Jiggles

Nervous or jiggly text apps are quite a common site on the Java scene. The Nervous Text applet, which is provided as part of the JDK, was one

Table 15.4 *Font Metric Information Methods*

Method	Description
getLeading	This is the appropriate amount of space to leave between lines of the font
getAscent	This is the distance from the baseline to the top of a character
getDescent	This is the distance from the baseline to the bottom of a character
getHeight	This is the distance from the top to the bottom of a character
getMaxAscent	This is the maximum ascent for all characters in the font
getMaxDecent	This is the maximum descent for all characters in the font
getMaxAdvance	This is the maximum space before any character in the font
charWidth	This is the width of an individual character
stringWidth	This is the width of an entire string

of the original look-what-I-can-do-Ma applets, that showed how Java could be used to add some eye-catching interest to an HTML page. Other applets have since taken Nervous Text and expanded its capabilities. I figured I could not write a section on text handling without showing you how it is done.

We have covered all the stuff we need to understand how it all works. Nervous texts are simply graphics areas with text being repeatedly repainted. The position of the text is controlled by randomly adjusting the font metric information, such as the height position of the text. At each repaint, the position is altered slightly.

Listing 15.25 shows a version of the **paint** method of a jiggly text applet. It assumes that the text is in a variable called **text**. I have also placed information, such as getting the **FontMetrics** object, in the **paint** method. Typically, though, this information does not change between repaints, so it would have been placed in the constructor of the applet.

Listing 15.25 Code for Jiggly Text

```
public void paint(Graphics g) {
    FontMetrics fm = getFontMetrics(getFont());
    char chars[] = new char[text.length()];
    text.getChars(0, text.length(), chars, 0);
    int positions [] = new int[text.length()];
    for (int i = 0; i < text.length(); i++) {
        positions[i] = fm.charsWidth(chars, 0, i);
    }

    for (int i = 0; i < text.length(); i++) {
        int x = (int)(Math.random() * jiggleFactor) + positions[i];
        int y = (int)(Math.random() * jiggleFactor) + fm.getAscent() - 1;
        g.drawChars(chars, i, 1, x, y);
    }
}
```

LET'S REVIEW

We have covered a lot in this chapter, but it has all centered around the **Graphics** class. The **Graphics** class is the core class you need to know if you want to draw graphics. The **Graphics** class is supported in its work by other classes such as the **Font**, **Canvas**, and the **Color** classes, to name but a few. As a whole these classes give you full control over the drawing you want to do.

We have not finished with the **Graphics** class. It is so central to all graphics related stuff that it will be popping up from time to time as a star player in other chapters.